ИНТЕРНЕТ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Основы операционных систем

Курс лекций. Учебное пособие
Издание второе, дополненное и исправленное

В.Е. Карпов
К.А. Коньков
под редакцией члена-корреспондента РАН
В.П. Иванникова

Рекомендовано для студентов высших учебных заведений, обучающихся по специальностям в области информационных технологий

Серия «Основы информационных технологий»

Курс разработан при поддержке компании Intel

Интернет-Университет Информационных Технологий
www.intuit.ru
Москва, 2005

УДК 004.451(075.8)
ББК 32.973.26-018.2я73-2
К26

К-26 Основы операционных систем. Курс лекций. Учебное пособие / В.Е. Карпов, К.А. Коньков / Под редакцией В.П. Иванникова. — М.: ИНТУИТ.РУ «Интернет-Университет Информационных Технологий», 2005. — 536 с. ISBN 5-9556-0044-2

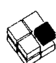
Книга представляет собой систематизированный учебный курс по теории операционных систем. В ней рассмотрены фундаментальные принципы построения и особенности проектирования современных ОС. Теоретический материал дополнен разнообразными практическими примерами.

Рекомендовано для студентов высших учебных заведений, обучающихся по специальностям в области информационных технологий.

Библиогр. 19

Курс разработан при поддержке компании Intel.

Издание осуществлено при финансовой и технической поддержке
издательства «Открытые Системы», «РМ Телеком» и Kraftway Computers.



ОТКРЫТЫЕ СИСТЕМЫ РМ Телеком

kraftway

Полное или частичное воспроизведение или размножение каким-либо способом, в том числе и публикация в Сети, настоящего издания допускается только с письменного разрешения Интернет-Университета Информационных Технологий.

© Интернет-Университет Информационных Технологий, www.intuit.ru, 2005

ISBN 5-9556-0044-2

О проекте

Интернет-Университет Информационных Технологий – это первое в России высшее учебное заведение, которое предоставляет возможность получить дополнительное образование во Всемирной сети. Web-сайт университета находится по адресу www.intuit.ru.

Мы рады, что вы решили расширить свои знания в области компьютерных технологий. Современный мир – это мир компьютеров и информации. Компьютерная индустрия – самый быстрорастущий сектор экономики, и ее рост будет продолжаться еще долгое время. Во времена жесткой конкуренции от уровня развития информационных технологий, достижений научной мысли и перспективных инженерных решений зависит успех не только отдельных людей и компаний, но и целых стран. Вы выбрали самое подходящее время для изучения компьютерных дисциплин. Профессионалы в области информационных технологий сейчас востребованы везде: в науке, экономике, образовании, медицине и других областях, в государственных и частных компаниях, в России и за рубежом. Анализ данных, прогнозы, организация связи, создание программного обеспечения, построение моделей процессов – вот далеко не полный список областей применения знаний для компьютерных специалистов.

Обучение в университете ведется по собственным учебным планам, разработанным ведущими российскими специалистами на основе международных образовательных стандартов Computer Curricula 2001 Computer Science. Изучать учебные курсы можно самостоятельно по учебникам или на сайте Интернет-университета, задания выполняются только на сайте. Для обучения необходимо зарегистрироваться на сайте университета. Удостоверение об окончании учебного курса или специальности выдается при условии выполнения всех заданий к лекциям и успешной сдачи итогового экзамена.

Книга, которую вы держите в руках, очередная в многотомной серии «Основы информационных технологий», выпускаемой Интернет-Университетом Информационных Технологий. В этой серии будут выпущены учебники по всем базовым областям знаний, связанным с компьютерными дисциплинами.

Добро пожаловать в Интернет-Университет Информационных Технологий!

Анатолий Шкред
anatoli@shkred.ru

Предисловие

Операционные системы (ОС) появились в учебных университетских курсах (вначале фрагментарно) в 1960-х годах. Например, в курсе по системному программированию, прочитанном Н. Виртом в Стенфордском университете. В 1970-х ОС становится одной из самостоятельных дисциплин в разделе информатики.

В зависимости от специализации курс операционных систем занимает один или несколько семестров. На сегодня издано уже множество книг по ОС. Среди них хотелось бы выделить две: «Операционные системы», Д. Цикритис и Ф. Бернстайн, изданную в 1977 году и широко использовавшуюся в преподавании и самостоятельном изучении на протяжении многих лет, и великолепную недавнюю монографию «Современные операционные системы» Э. Таненбаума.

Книга, предисловие к которой вы сейчас читаете, необычна по форме. Она содержит две параллельные нити изложения: лекции и практикум-семинар. В лекционном курсе даются фундаментальные знания по ОС. Они не привязаны к конкретной ОС. Практикум-семинар – это своеобразная синтетическая форма семинаров, которые проводятся преподавателем в компьютерном классе одновременно с выполнением студентами заданий, привязанных к конкретной ОС. Безусловно, лекции и практикум-семинары строго связаны между собой. Практикум-семинар является сменной нитью и в данном издании это Unix.

В основу книги положен курс ОС МФТИ третьего семестра, читаемый в течение последних пяти лет. Ему предшествуют два семестровых курса: «Введение в алгоритмы» и «Введение в архитектуру вычислительных систем». Знания и навыки, приобретаемые студентами за первые два семестра, используются в курсе «Введение в операционные системы». Наиболее важными являются структуры данных, в особенности организация очередей и таблиц, оценка сложности алгоритмов (первый семестр); организация памяти, система прерываний, машинные инструкции и отображение конструкций языков программирования в систему команд (второй семестр).

По существу, эта книга представляет собой учебник, который можно непосредственно использовать в программах университетов и рекомендовать как лекторам, так и студентам. Достаточно хорошая библиография по всем разделам позволяет модифицировать курс. Однако делать это нужно одновременно с коррекцией практикума-семинара. Книга может использоваться и для самостоятельного изучения основ операционных систем.

В.П. Иванников,
профессор, член-корреспондент РАН

Об авторах

Карпов Владимир Ефимович

Кандидат физико-математических наук, доцент кафедры информатики Московского физико-технического института, сопредседатель секции «Компьютеры в науке и образовании» ежегодной международной конференции «Математика, компьютер, образование». Сотрудничал в разработке разнообразных программных систем с компаниями Paralogic, AT&T. Автор ряда статей и международных докладов по параллельному программированию и распараллеливанию задач математической физики.

Коньков Константин Алексеевич

Кандидат физико-математических наук, старший научный сотрудник Института Автоматизации проектирования РАН, доцент кафедры математических и информационных технологий Московского физико-технического института, директор по научной работе ООО «Физтех-софт». Область научных интересов: операционные системы, разработка системного программного обеспечения. Принимал участие в создании операционной системы PTS-DOS, системы защиты информации StrongDisk и ряда других системных продуктов.

Лекции

Лекция 1. Введение 13
Лекция 2. Процессы 41
Лекция 3. Планирование процессов 59
Лекция 4. Кооперация процессов и основные аспекты ее логической организации 87
Лекция 5. Алгоритмы синхронизации 105
Лекция 6. Механизмы синхронизации 123
Лекция 7. Тупики 141
Лекция 8. Организация памяти компьютера. Простейшие схемы управления памятью 155
Лекция 9. Виртуальная память. Архитектурные средства поддержки виртуальной памяти 175
Лекция 10. Аппаратно-независимый уровень управления виртуальной памятью 189
Лекция 11. Файлы с точки зрения пользователя 209
Лекция 12. Реализация файловой системы 229
Лекция 13. Система управления вводом-выводом 265
Лекция 14. Сети и сетевые операционные системы 295
Лекция 15. Основные понятия информационной безопасности 325
Лекция 16. Защитные механизмы операционных систем 339

Семинары 1-2. Введение в курс практических занятий. Знакомство с операционной системой UNIX.	Основываются на лекции 1
Семинары 3-4. Процессы в операционной системе UNIX.	Основываются на лекции 2
Семинар 5. Организация взаимодействия процессов через pipe и FIFO в UNIX.	Основывается на лекции 4
Семинары 6-7. Средства System V IPC. Организация работы с разделяемой памятью в UNIX. Понятие нитей исполнения (thread).	Основываются на лекциях 4 и 5
Семинар 8. Семафоры в UNIX как средство синхронизации процессов.	Основывается на лекциях 5 и 6
Семинар 9. Очереди сообщений в UNIX.	Основывается на лекциях 4 и 6
Семинары 10-11. Организация файловой системы в UNIX. Работа с файлами и директориями. Понятие о memory mapped файлах.	Основываются на лекциях 10, 11 и 12
Семинары 12-13. Организация ввода-вывода в UNIX. Файлы устройств. Аппарат прерываний. Сигналы в UNIX.	Основываются на лекциях 12 и 13
Семинары 14-15. Семейство протоколов TCP/IP. Сокеты (sockets) в UNIX и основы работы с ними.	Основываются на лекции 14

Содержание

Часть I. Обзор 13
Лекция 1. Введение 13
Что такое операционная система 13
Краткая история эволюции вычислительных систем 16
Основные понятия, концепции ОС 24
Архитектурные особенности ОС 27
Классификация ОС 32
Заключение 34
Часть II. Процессы и их поддержка в операционной системе 41
Лекция 2. Процессы 41
Понятие процесса 41
Состояния процесса 43
Операции над процессами и связанные с ними понятия 46
Заключение 54
Лекция 3. Планирование процессов 59
Уровни планирования 59
Критерии планирования и требования к алгоритмам 61
Параметры планирования 62
Вытесняющее и невытесняющее планирование 64
Алгоритмы планирования 65
Заключение 82
Лекция 4. Кооперация процессов и основные аспекты ее логической организации 87
Взаимодействующие процессы 87
Категории средств обмена информацией 89
Логическая организация механизма передачи информации 90
Нити исполнения 96
Заключение 100
Лекция 5. Алгоритмы синхронизации 105
Interleaving, race condition и взаимоисключения 105
Критическая секция 108
Программные алгоритмы организации взаимодействия процессов 111
Аппаратная поддержка взаимоисключений 117
Заключение 119
Лекция 6. Механизмы синхронизации 123
Семафоры 123
Мониторы 126
Сообщения 129
Эквивалентность семафоров, мониторов и сообщений 130

Заключение . . . 133
Лекция 7. Тупики . . . 141
Введение . . . 141
Условия возникновения тупиков . . . 143
Основные направления борьбы с тупиками . . . 143
Игнорирование проблемы тупиков . . . 144
Способы предотвращения тупиков . . . 144
Обнаружение тупиков . . . 149
Восстановление после тупиков . . . 150
Заключение . . . 151
Часть III. Управление памятью . . . 155
Лекция 8. Организация памяти компьютера. Простейшие схемы управления памятью . . . 155
Введение . . . 155
Простейшие схемы управления памятью . . . 161
Страничная память . . . 166
Сегментная и сегментно-страничная организация памяти . . . 168
Заключение . . . 171
Лекция 9. Виртуальная память. Архитектурные средства поддержки виртуальной памяти . . . 175
Понятие виртуальной памяти . . . 175
Архитектурные средства поддержки виртуальной памяти . . . 177
Заключение . . . 185
Лекция 10. Аппаратно-независимый уровень управления виртуальной памятью . . . 189
Исключительные ситуации при работе с памятью . . . 189
Стратегии управления страничной памятью . . . 190
Алгоритмы замещения страниц . . . 191
Управление количеством страниц, выделенным процессу. Модель рабочего множества . . . 197
Страничные демоны . . . 200
Программная поддержка сегментной модели памяти процесса . . 201
Отдельные аспекты функционирования менеджера памяти . . . 203
Заключение . . . 205
Часть IV. Файловые системы . . . 209
Лекция 11. Файлы с точки зрения пользователя . . . 209
Введение . . . 209
Общие сведения о файлах . . . 212
Организация файлов и доступ к ним . . . 214
Операции над файлами . . . 217
Директории. Логическая структура файлового архива . . . 218
Операции над директориями . . . 222

Защита файлов 223
Заключение 224
Лекция 12. Реализация файловой системы 229
Общая структура файловой системы 229
Управление внешней памятью 232
Реализация директорий 240
Монтирование файловых систем 243
Связывание файлов 245
Кооперация процессов при работе с файлами 247
Надежность файловой системы 250
Производительность файловой системы 254
Реализация некоторых операций над файлами. 256
Современные архитектуры файловых систем 259
Заключение 260
Часть V. Ввод-вывод 265
Лекция 13. Система управления вводом-выводом 265
Физические принципы организации ввода-вывода 266
Логические принципы организации ввода-вывода. 276
Алгоритмы планирования запросов к жесткому диску 286
Заключение 291
Часть VI. Сети и сетевые операционные системы. 295
Лекция 14. Сети и сетевые операционные системы 295
Для чего компьютеры объединяют в сети 296
Сетевые и распределенные операционные системы. 297
Взаимодействие удаленных процессов как основа работы вычислительных сетей 298
Основные вопросы логической организации передачи информации между удаленными процессами 301
Понятие протокола. 302
Многоуровневая модель построения сетевых вычислительных систем. 304
Проблемы адресации в сети 308
Проблемы маршрутизации в сетях 315
Связь с установлением логического соединения и передача данных с помощью сообщений 318
Синхронизация удаленных процессов 320
Заключение 320
Часть VII. Проблемы безопасности операционных систем 325
Лекция 15. Основные понятия информационной безопасности. . . . 325
Введение 325
Угрозы безопасности 327

Формализация подхода к обеспечению информационной безопасности 329
Криптография как одна из базовых технологий безопасности ОС 331
Заключение 335
Лекция 16. Защитные механизмы операционных систем 339
Идентификация и аутентификация 339
Авторизация. Разграничение доступа к объектам ОС 342
Выявление вторжений. Аудит системы защиты 347
Анализ некоторых популярных ОС с точки зрения их защищенности 348
Заключение 353

Семинары 357
Семинары 1–2. Введение в курс практических занятий. Знакомство с операционной системой UNIX 357
Семинары 3–4. Процессы в операционной системе UNIX 385
Семинар 5. Организация взаимодействия процессов через pipe и FIFO в UNIX 403
Семинары 6–7. Средства System V IPC. Организация работы с разделяемой памятью в UNIX. Понятие нитей исполнения (thread) 431
Семинар 8. Семафоры в UNIX как средство синхронизации процессов 465
Семинар 9. Очереди сообщений в UNIX 479
Семинары 10–11. Организация файловой системы в UNIX. Работа с файлами и директориями. Понятие о memory mapped файлах 501
Семинары 12–13. Организация ввода-вывода в UNIX. Файлы устройств. Аппарат прерываний. Сигналы в UNIX 539
Семинары 14–15. Семейство протоколов TCP/IP. Сокеты (sockets) в UNIX и основы работы с ними 575

Литература 627

Часть I. Обзор

Лекция 1. Введение

В данной лекции вводится понятие операционной системы; рассматривается эволюция развития операционных систем; описываются функции операционных систем и подходы к построению операционных систем.

Ключевые слова: операционная система (ОС), виртуальная машина, менеджер ресурсов, лампа, транзисторы, интегральные схемы, микропроцессоры, пакетная система, многозадачная ОС, ОС с разделением времени, многопользовательская ОС, сетевая ОС, распределенная ОС, прерывание, системный вызов, исключительная ситуация, монолитное ядро, микроядерная архитектура, многопроцессорная ОС, система реального времени.

Операционная система (ОС) – это программа, которая обеспечивает возможность рационального использования оборудования компьютера удобным для пользователя образом. Вводная лекция рассказывает о предмете, изучаемом в рамках настоящего курса. Вначале мы попытаемся ответить на вопрос, что такое ОС. Затем последует анализ эволюции ОС и рассказ о возникновении основных концепций и компонентов современных ОС. В заключение будет представлена классификация ОС с точки зрения особенностей архитектуры и использования ресурсов компьютера.

Что такое операционная система

Структура вычислительной системы

Из чего состоит любая вычислительная система? Во-первых, из того, что в англоязычных странах принято называть словом hardware, или техническое обеспечение: процессор, память, монитор, дисковые устройства и т. д., объединенные магистральным соединением, которое называется шиной. Некоторые сведения об архитектуре компьютера имеются в приложении 1 к настоящей лекции.

Во-вторых, вычислительная система состоит из *программного обеспечения*. Все программное обеспечение принято делить на две части: прикладное и системное. К прикладному программному обеспечению, как правило, относятся разнообразные банковские и прочие бизнес-програм-

мы, игры, текстовые процессоры и т. п. Под системным программным обеспечением обычно понимают программы, способствующие функционированию и разработке прикладных программ. Надо сказать, что деление на прикладное и системное программное обеспечение является отчасти условным и зависит от того, кто осуществляет такое деление. Так, обычный пользователь, неискушенный в программировании, может считать Microsoft Word системной программой, а, с точки зрения программиста, это – приложение. Компилятор языка Си для обычного программиста – системная программа, а для системного – прикладная. Несмотря на эту нечеткую грань, данную ситуацию можно отобразить в виде последовательности слоев (см. рис 1.1), выделив отдельно наиболее общую часть системного программного обеспечения — операционную систему:

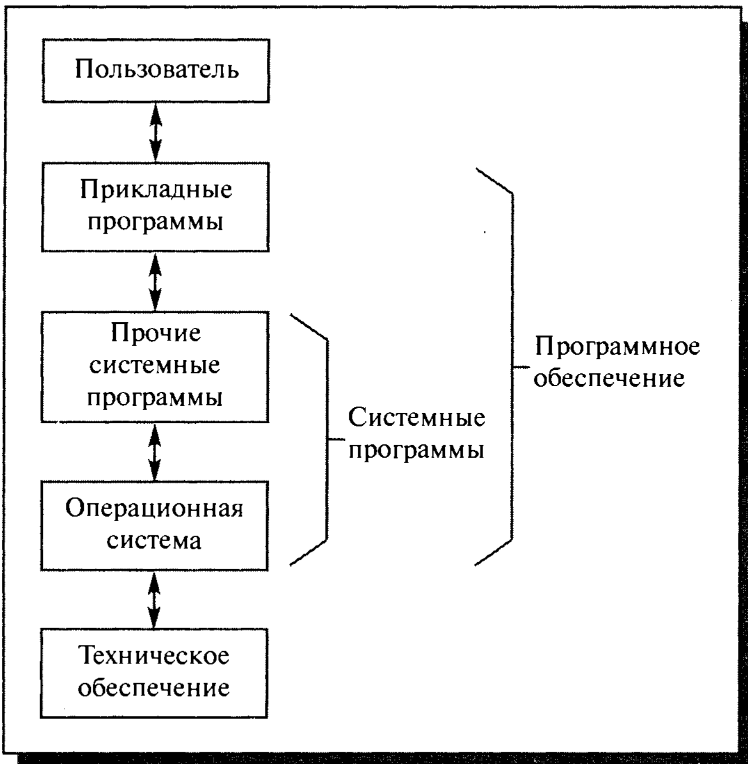


Рис. 1.1. Слои программного обеспечения компьютерной системы

Что такое ОС

Большинство пользователей имеет опыт эксплуатации операционных систем, но тем не менее они затруднятся дать этому понятию точное определение. Давайте кратко рассмотрим основные точки зрения.

Операционная система как виртуальная машина

При разработке ОС широко применяется абстрагирование, которое является важным методом упрощения и позволяет сконцентрироваться на взаимодействии высокоуровневых компонентов системы, игнорируя детали их реализации. В этом смысле ОС представляет собой интерфейс между пользователем и компьютером.

Архитектура большинства компьютеров на уровне машинных команд очень неудобна для использования прикладными программами. Например, работа с диском предполагает знание внутреннего устройства его электронного компонента – контроллера для ввода команд вращения диска, поиска и форматирования дорожек, чтения и записи секторов и т. д. Ясно, что средний программист не в состоянии учитывать все особенности работы оборудования (в современной терминологии – заниматься разработкой драйверов устройств), а должен иметь простую высокоуровневую абстракцию, скажем, представляя информационное пространство диска как набор файлов. Файл можно открывать для чтения или записи, использовать для получения или сброса информации, а потом закрывать. Это концептуально проще, чем заботиться о деталях перемещения головок дисков или организации работы мотора. Аналогичным образом, с помощью простых и ясных абстракций, скрываются от программиста все ненужные подробности организации прерываний, работы таймера, управления памятью и т. д. Более того, на современных вычислительных комплексах можно создать иллюзию неограниченного размера операционной памяти и числа процессоров. Всем этим занимается операционная система. Таким образом, операционная система представляется пользователю виртуальной машиной, с которой проще иметь дело, чем непосредственно с оборудованием компьютера.

Операционная система как менеджер ресурсов

Операционная система предназначена для управления всеми частями весьма сложной архитектуры компьютера. Представим, к примеру, что произойдет, если несколько программ, работающих на одном компьютере, будут пытаться одновременно осуществлять вывод на принтер. Мы получили бы мешанину строчек и страниц, выведенных различными

программами. Операционная система предотвращает такого рода хаос за счет буферизации информации, предназначенной для печати, на диске и организации очереди на печать. Для многопользовательских компьютеров необходимость управления ресурсами и их защиты еще более очевидна. Следовательно, операционная система, как менеджер ресурсов, осуществляет упорядоченное и контролируемое распределение процессоров, памяти и других ресурсов между различными программами.

Операционная система как защитник пользователей и программ

Если вычислительная система допускает совместную работу нескольких пользователей, то возникает проблема организации их безопасной деятельности. Необходимо обеспечить сохранность информации на диске, чтобы никто не мог удалить или повредить чужие файлы. Нельзя разрешить программам одних пользователей произвольно вмешиваться в работу программ других пользователей. Нужно пресекать попытки несанкционированного использования вычислительной системы. Всю эту деятельность осуществляет операционная система как организатор безопасной работы пользователей и их программ. С такой точки зрения операционная система представляется системой безопасности государства, на которую возложены полицейские и контрразведывательные функции.

Операционная система как постоянно функционирующее ядро

Наконец, можно дать и такое определение: операционная система – это программа, постоянно работающая на компьютере и взаимодействующая со всеми прикладными программами. Казалось бы, это абсолютно правильное определение, но, как мы увидим дальше, во многих современных операционных системах постоянно работает на компьютере лишь часть операционной системы, которую принято называть ее ядром.

Как мы видим, существует много точек зрения на то, что такое операционная система. Невозможно дать ей адекватное строгое определение. Нам проще сказать не что есть операционная система, а для чего она нужна и что она делает. Для выяснения этого вопроса рассмотрим историю развития вычислительных систем.

Краткая история эволюции вычислительных систем

Мы будем рассматривать историю развития именно вычислительных, а не операционных систем, потому что hardware и программное обеспечение эволюционировали совместно, оказывая взаимное влияние

друг на друга. Появление новых технических возможностей приводило к прорыву в области создания удобных, эффективных и безопасных программ, а свежие идеи в программной области стимулировали поиски новых технических решений. Именно эти критерии – удобство, эффективность и безопасность – играли роль факторов естественного отбора при эволюции вычислительных систем.

Первый период (1945-1955 гг.). Ламповые машины. Операционных систем нет

Мы начнем исследование развития компьютерных комплексов с появления электронных вычислительных систем (опуская историю механических и электромеханических устройств).

Первые шаги в области разработки электронных вычислительных машин были предприняты в конце Второй Мировой войны. В середине 1940-х годов были созданы первые ламповые вычислительные устройства и появился принцип программы, хранящейся в памяти машины (John Von Neumann, июнь 1945 г.). В то время одна и та же группа людей участвовала и в проектировании, и в эксплуатации, и в программировании вычислительной машины. Это была скорее научно-исследовательская работа в области вычислительной техники, а не регулярное использование компьютеров в качестве инструмента решения каких-либо практических задач из других прикладных областей. Программирование осуществлялось исключительно на машинном языке. Об операционных системах не было и речи, все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления. За пультом мог находиться только один пользователь. Программа загружалась в память машины в лучшем случае с колоды перфокарт, а обычно — с помощью панели переключателей.

Вычислительная система выполняла одновременно только одну операцию (ввод-вывод или собственно вычисления). Отладка программ велась с пульта управления с помощью изучения состояния памяти и регистров машины. В конце этого периода появляется первое системное программное обеспечение: в 1951-1952 гг. возникают прообразы первых компиляторов с символических языков (Fortran и др.), а в 1954 г. Nat Rochester разрабатывает Ассемблер для IBM-701.

Существенная часть времени уходила на подготовку запуска программы, а сами программы выполнялись строго последовательно. Такой режим работы называется *последовательной обработкой* данных. В целом первый период характеризуется крайне высокой стоимостью вычислительных систем, их малым количеством и низкой эффективностью использования.

Второй период (1955 г.–начало 1960-х гг.). Компьютеры на основе транзисторов. Пакетные операционные системы

С середины 1950-х годов начался следующий период в эволюции вычислительной техники, связанный с появлением новой технической базы – полупроводниковых элементов. Применение транзисторов вместо часто перегоравших электронных ламп привело к повышению надежности компьютеров. Теперь машины могли непрерывно работать достаточно долго, чтобы на них можно было возложить выполнение практически важных задач. Снизилось потребление вычислительными машинами электроэнергии, усовершенствовались системы охлаждения. Размеры компьютеров уменьшились. Снизилась стоимость эксплуатации и обслуживания вычислительной техники. Началось использование ЭВМ коммерческими фирмами. Одновременно наблюдается бурное развитие алгоритмических языков (LISP, COBOL, ALGOL-60, PL-1 и т.д.). Появляются первые настоящие компиляторы, редакторы связей, библиотеки математических и служебных подпрограмм. Упрощается процесс программирования. Пропадает необходимость взваливать на одних и тех же людей весь процесс разработки и использования компьютеров. Именно в этот период происходит разделение персонала на программистов и операторов, разработчиков вычислительных машин и специалистов по эксплуатации.

Изменяется сам процесс прогона программ. Теперь пользователь приносит программу с входными данными в виде колоды перфокарт и указывает необходимые ресурсы. Такая колода получает название *задания*. Оператор загружает задание в память машины и запускает его на исполнение. Полученные выходные данные печатаются на принтере, и пользователь получает их обратно через некоторое (довольно продолжительное) время.

Смена запрошенных ресурсов вызывает приостановку выполнения программ, в результате процессор часто простаивает. Для повышения эффективности использования компьютера задания с похожими ресурсами начинают собирать вместе, создавая *пакет заданий*.

Появляются первые системы *пакетной обработки*, которые просто автоматизируют запуск одной программы из пакета за другой и тем самым увеличивают коэффициент загрузки процессора. При реализации систем пакетной обработки был разработан формализованный язык управления заданиями, с помощью которого программист сообщал системе и оператору, какую работу он хочет выполнить на вычислительной машине. Системы пакетной обработки стали прообразом современных операционных систем, они были первыми системными программами, предназначенными для управления вычислительным процессом.

Третий период (начало 1960-х – 1980 г.). Компьютеры на основе интегральных микросхем. Первые многозадачные ОС

Следующий важный период развития вычислительных машин относится к началу 1960-х – 1980 г. В это время в технической базе произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам. Вычислительная техника становится более надежной и дешевой. Растет сложность и количество задач, решаемых компьютерами. Повышается производительность процессоров.

Повышению эффективности использования процессорного времени мешает низкая скорость работы механических устройств ввода-вывода (быстрый считыватель перфокарт мог обработать 1200 перфокарт в минуту, принтеры печатали до 600 строк в минуту). Вместо непосредственного чтения пакета заданий с перфокарт в память, начинают использовать его предварительную запись, сначала на магнитную ленту, а затем и на диск. Когда в процессе выполнения задания требуется ввод данных, они читаются с диска. Точно так же выходная информация сначала копируется в системный буфер и записывается на ленту или диск, а печатается только после завершения задания. Вначале действительные операции ввода-вывода осуществлялись в режиме off-line, то есть с использованием других, более простых, отдельно стоящих компьютеров. В дальнейшем они начинают выполняться на том же компьютере, который производит вычисления, то есть в режиме on-line. Такой прием получает название spooling (сокращение от Simultaneous Peripheral Operation On Line) или подкачки-откачки данных. Введение техники подкачки-откачки в пакетные системы позволило совместить реальные операции ввода-вывода одного задания с выполнением другого задания, но потребовало разработки аппарата прерываний для извещения процессора об окончании этих операций.

Магнитные ленты были устройствами последовательного доступа, то есть информация считывалась с них в том порядке, в каком была записана. Появление магнитного диска, для которого не важен порядок чтения информации, то есть устройства прямого доступа, привело к дальнейшему развитию вычислительных систем. При обработке пакета заданий на магнитной ленте очередность запуска заданий определялась порядком их ввода. При обработке пакета заданий на магнитном диске появилась возможность выбора очередного выполняемого задания. Пакетные системы начинают заниматься *планированием заданий*: в зависимости от наличия запрошенных ресурсов, срочности вычислений и т. д. на счет выбирается то или иное задание.

Дальнейшее повышение эффективности использования процессора было достигнуто с помощью *мультипрограммирования*. Идея мультипрограммирования заключается в следующем: пока одна программа выполня-

ет операцию ввода-вывода, процессор не простаивает, как это происходило при однопрограммном режиме, а выполняет другую программу. Когда операция ввода-вывода заканчивается, процессор возвращается к выполнению первой программы. Эта идея напоминает поведение преподавателя и студентов на экзамене. Пока один студент (программа) обдумывает ответ на вопрос (операция ввода-вывода), преподаватель (процессор) выслушивает ответ другого студента (вычисления). Естественно, такая ситуация требует наличия в комнате нескольких студентов. Точно так же мультипрограммирование требует наличия в памяти нескольких программ одновременно. При этом каждая программа загружается в свой участок оперативной памяти, называемый разделом, и не должна влиять на выполнение другой программы. (Студенты сидят за отдельными столами и не подсказывают друг другу.)

Появление мультипрограммирования требует настоящей революции в строении вычислительной системы. Особую роль здесь играет аппаратная поддержка (многие аппаратные новшества появились еще на предыдущем этапе эволюции), наиболее существенные особенности которой перечислены ниже.

- Реализация защитных механизмов. Программы не должны иметь самостоятельного доступа к распределению ресурсов, что приводит к появлению *привилегированных* и *непривилегированных команд*. Привилегированные команды, например команды ввода-вывода, могут исполняться только операционной системой. Говорят, что она работает в привилегированном режиме. Переход управления от прикладной программы к ОС сопровождается контролируемой сменой режима. Кроме того, это защита памяти, позволяющая изолировать конкурирующие пользовательские программы друг от друга, а ОС – от программ пользователей.
- Наличие прерываний. Внешние прерывания оповещают ОС о том, что произошло асинхронное событие, например завершилась операция ввода-вывода. Внутренние прерывания (сейчас их принято называть исключительными ситуациями) возникают, когда выполнение программы привело к ситуации, требующей вмешательства ОС, например деление на ноль или попытка нарушения защиты.
- Развитие параллелизма в архитектуре. Прямой доступ к памяти и организация каналов ввода-вывода позволили освободить центральный процессор от рутинных операций.

Не менее важна в организации мультипрограммирования роль операционной системы. Она отвечает за следующие операции:

- Организация интерфейса между прикладной программой и ОС при помощи *системных вызовов*.

- Организация очереди из заданий в памяти и выделение процессора одному из заданий требует *планирования* использования процессора.
- Переключение с одного задания на другое требует сохранения содержимого регистров и структур данных, необходимых для выполнения задания, иначе говоря, *контекста* для обеспечения правильного продолжения вычислений.
- Поскольку память является ограниченным ресурсом, нужны *стратегии управления памятью*, то есть требуется упорядочить процессы размещения, замещения и выборки информации из памяти.
- Организация хранения информации на внешних носителях в виде файлов и обеспечение доступа к конкретному файлу только определенным категориям пользователей.
- Поскольку программам может потребоваться произвести санкционированный обмен данными, необходимо их обеспечить *средствами коммуникации*.
- Для корректного обмена данными необходимо разрешать конфликтные ситуации, возникающие при работе с различными ресурсами, и предусмотреть координацию программами своих действий, т. е. снабдить систему *средствами синхронизации*.

Мультипрограммные системы обеспечили возможность более эффективного использования системных ресурсов (например, процессора, памяти, периферийных устройств), но они еще долго оставались пакетными. Пользователь не мог непосредственно взаимодействовать с заданием и должен был предусмотреть с помощью управляющих карт все возможные ситуации. Отладка программ по-прежнему занимала много времени и требовала изучения многостраничных распечаток содержимого памяти и регистров или использования отладочной печати.

Появление электронно-лучевых дисплеев и переосмысление возможностей применения клавиатур поставили на очередь решение этой проблемы. Логическим расширением систем мультипрограммирования стали *time-sharing* системы, или *системы разделения времени* *. В них процессор переключается между задачами не только на время операций ввода-вывода, но и просто по прошествии определенного времени. Эти переключения происходят так часто, что пользователи могут взаимодействовать со своими программами во время их выполнения, то есть интерактивно. В результате появляется возможность одновременной работы нескольких пользователей на одной компьютерной системе. У каждого пользователя для этого должна быть хотя бы одна программа в памяти.

* Реальная эволюция операционных систем происходила не так гладко и планомерно, как это представлено в данном обзоре. Так, например, первая система с разделением времени Joss была реализована еще на ламповой машине Joniac безо всякой аппаратной поддержки.

Чтобы уменьшить ограничения на количество работающих пользователей, была внедрена идея неполного нахождения исполняемой программы в оперативной памяти. Основная часть программы находится на диске и фрагмент, который необходимо в данный момент выполнять, может быть загружен в оперативную память, а ненужный – выкачан обратно на диск. Это реализуется с помощью механизма *виртуальной памяти*. Основным достоинством такого механизма является создание иллюзии неограниченной оперативной памяти ЭВМ.

В системах разделения времени пользователь получил возможность эффективно производить отладку программы в интерактивном режиме и записывать информацию на диск, не используя перфокарты, а непосредственно с клавиатуры. Появление on-line-файлов привело к необходимости разработки развитых *файловых систем*.

Параллельно внутренней эволюции вычислительных систем происходила и внешняя их эволюция. До начала этого периода вычислительные комплексы были, как правило, несовместимы. Каждый имел собственную операционную систему, свою систему команд и т. д. В результате программу, успешно работающую на одном типе машин, необходимо было полностью переписывать и заново отлаживать для выполнения на компьютерах другого типа. В начале третьего периода появилась идея создания семейств программно совместимых машин, работающих под управлением одной и той же операционной системы. Первым семейством программно совместимых компьютеров, построенных на интегральных микросхемах, стала серия машин IBM/360. Разработанное в начале 1960-х годов, это семейство значительно превосходило машины второго поколения по критерию цена/производительность. За ним последовала линия компьютеров PDP, несовместимых с линией IBM, и лучшей моделью в ней стала PDP-11.

Сила «одной семьи» была одновременно и ее слабостью. Широкие возможности этой концепции (наличие всех моделей: от мини-компьютеров до гигантских машин; обилие разнообразной периферии; различное окружение; различные пользователи) порождали сложную и громоздкую операционную систему. Миллионы строчек Ассемблера, написанные тысячами программистов, содержали множество ошибок, что вызывало непрерывный поток публикаций о них и попыток исправления. Только в операционной системе OS/360 содержалось более 1000 известных ошибок. Тем не менее идея стандартизации операционных систем была широко внедрена в сознание пользователей и в дальнейшем получила активное развитие.

Четвертый период (с 1980 г. по настоящее время) Персональные компьютеры. Классические, сетевые и распределенные системы

Следующий период в эволюции вычислительных систем связан с появлением больших интегральных схем (БИС). В эти годы произошло резкое возрастание степени интеграции и снижение стоимости микросхем. Компьютер, не отличающийся по архитектуре от PDP-11, по цене и простоте эксплуатации стал доступен отдельному человеку, а не только отделу предприятия или университета. Наступила эра персональных компьютеров. Первоначально персональные компьютеры предназначались для использования одним пользователем в однопрограммном режиме, что повлекло за собой деградацию архитектуры этих ЭВМ и их операционных систем (в частности, пропала необходимость защиты файлов и памяти, планирования заданий и т. п.).

Компьютеры стали использоваться не только специалистами, что потребовало разработки «дружественного» программного обеспечения.

Однако рост сложности и разнообразия задач, решаемых на персональных компьютерах, необходимость повышения надежности их работы привели к возрождению практически всех черт, характерных для архитектуры больших вычислительных систем.

В середине 1980-х годов стали бурно развиваться сети компьютеров, в том числе персональных, работающих под управлением *сетевых* или *распределенных* операционных систем.

В сетевых операционных системах пользователи могут получить доступ к ресурсам другого сетевого компьютера, только они должны знать об их наличии и уметь это сделать. Каждая машина в сети работает под управлением своей локальной операционной системы, отличающейся от операционной системы автономного компьютера наличием дополнительных средств (программной поддержкой для сетевых интерфейсных устройств и доступа к удаленным ресурсам), но эти дополнения не меняют структуру операционной системы.

Распределенная система, напротив, внешне выглядит как обычная автономная система. Пользователь не знает и не должен знать, где хранятся его файлы – на локальной или удаленной машине – и где его программы выполняются. Он может вообще не знать, подключен ли его компьютер к сети. Внутреннее строение распределенной операционной системы имеет существенные отличия от автономных систем.

В дальнейшем автономные операционные системы мы будем называть *классическими* операционными системами.

Просмотрев этапы развития вычислительных систем, мы можем выделить шесть основных функций, которые выполняли классические операционные системы в процессе эволюции:

- Планирование заданий и использования процессора.
- Обеспечение программ средствами коммуникации и синхронизации.
- Управление памятью.
- Управление файловой системой.
- Управление вводом-выводом.
- Обеспечение безопасности.

Каждая из приведенных функций обычно реализована в виде подсистемы, являющейся структурным компонентом ОС. В каждой операционной системе эти функции, конечно, реализовывались по-своему, в различном объеме. Они не были изначально придуманы как составные части операционных систем, а появились в процессе развития, по мере того как вычислительные системы становились все более удобными, эффективными и безопасными. Эволюция вычислительных систем, созданных человеком, пошла по такому пути, но никто еще не доказал, что это единственно возможный путь их развития. Операционные системы существуют потому, что на данный момент их существование – это разумный способ использования вычислительных систем. Рассмотрение общих принципов и алгоритмов реализации их функций и составляет содержание большей части нашего курса, в котором будут последовательно описаны перечисленные подсистемы.

Основные понятия, концепции ОС

В процессе эволюции возникло несколько важных концепций, которые стали неотъемлемой частью теории и практики ОС. Рассматриваемые в данном разделе понятия будут встречаться и разъясняться на протяжении всего курса. Здесь дается их краткое описание.

Системные вызовы

В любой операционной системе поддерживается механизм, который позволяет пользовательским программам обращаться к услугам ядра ОС. В операционных системах наиболее известной советской вычислительной машины БЭСМ-6 соответствующие средства «общения» с ядром назывались экстракодами, в операционных системах IBM они назывались системными макрокомандами и т. д. В ОС Unix такие средства называют *системными вызовами*.

Системные вызовы (system calls) – это интерфейс между операционной системой и пользовательской программой. Они создают, удаляют и

используют различные объекты, главные из которых – *процессы* и *файлы.* Пользовательская программа запрашивает сервис у операционной системы, осуществляя системный вызов. Имеются библиотеки процедур, которые загружают машинные регистры определенными параметрами и осуществляют прерывание процессора, после чего управление передается обработчику данного вызова, входящему в ядро операционной системы. Цель таких библиотек – сделать системный вызов похожим на обычный вызов подпрограммы.

Основное отличие состоит в том, что при системном вызове задача переходит в привилегированный режим или *режим ядра (kernel mode).* Поэтому системные вызовы иногда еще называют программными прерываниями, в отличие от аппаратных прерываний, которые чаще называют просто прерываниями.

В этом режиме работает код ядра операционной системы, причем исполняется он в адресном пространстве и в контексте вызвавшей его задачи. Таким образом, ядро операционной системы имеет полный доступ к памяти пользовательской программы, и при системном вызове достаточно передать адреса одной или нескольких областей памяти с параметрами вызова и адреса одной или нескольких областей памяти для результатов вызова.

В большинстве операционных систем системный вызов осуществляется командой программного прерывания (INT). Программное прерывание – это *синхронное* событие, которое может быть повторено при выполнении одного и того же программного кода.

Прерывания

Прерывание (hardware interrupt) – это событие, генерируемое внешним (по отношению к процессору) устройством. Посредством аппаратных прерываний аппаратура либо информирует центральный процессор о том, что произошло какое-либо событие, требующее немедленной реакции (например, пользователь нажал клавишу), либо сообщает о завершении асинхронной операции ввода-вывода (например, закончено чтение данных с диска в основную память). Важный тип аппаратных прерываний – прерывания таймера, которые генерируются периодически через фиксированный промежуток времени. Прерывания таймера используются операционной системой при планировании процессов. Каждый тип аппаратных прерываний имеет собственный номер, однозначно определяющий источник прерывания. Аппаратное прерывание – это *асинхронное* событие, то есть оно возникает вне зависимости от того, какой код исполняется процессором в данный момент. Обработка аппаратного прерывания не должна учитывать, какой процесс является текущим.

Исключительные ситуации

Исключительная ситуация (exception) – событие, возникающее в результате попытки выполнения программой команды, которая по каким-то причинам не может быть выполнена до конца. Примерами таких команд могут быть попытки доступа к ресурсу при отсутствии достаточных привилегий или обращения к отсутствующей странице памяти. Исключительные ситуации, как и системные вызовы, являются синхронными событиями, возникающими в контексте текущей задачи. Исключительные ситуации можно разделить на исправимые и неисправимые. К исправимым относятся такие исключительные ситуации, как отсутствие нужной информации в оперативной памяти. После устранения причины исправимой исключительной ситуации программа может выполняться дальше. Возникновение в процессе работы операционной системы исправимых исключительных ситуаций считается нормальным явлением. Неисправимые исключительные ситуации чаще всего возникают в результате ошибок в программах (например, деление на ноль). Обычно в таких случаях операционная система реагирует завершением программы, вызвавшей исключительную ситуацию.

Файлы

Файлы предназначены для хранения информации на внешних носителях, то есть принято, что информация, записанная, например, на диске, должна находиться внутри файла. Обычно под файлом понимают именованную часть пространства на носителе информации.

Главная задача *файловой системы (file system)* – скрыть особенности ввода-вывода и дать программисту простую абстрактную модель файлов, независимых от устройств. Для чтения, создания, удаления, записи, открытия и закрытия файлов также имеется обширная категория системных вызовов (создание, удаление, открытие, закрытие, чтение и т. д.). Пользователям хорошо знакомы такие связанные с организацией файловой системы понятия, как каталог, текущий каталог, корневой каталог, путь. Для манипулирования этими объектами в операционной системе имеются системные вызовы. Файловая система ОС описана в лекциях 11-12.

Процессы, нити

Концепция процесса в ОС — одна из наиболее фундаментальных. Процессы подробно рассмотрены в лекциях 2-7. Там же описаны нити, или легковесные процессы.

Архитектурные особенности ОС

До сих пор мы говорили о взгляде на операционные системы извне, о том, что делают операционные системы. Дальнейший наш курс будет посвящен тому, как они это делают. Но мы пока ничего не сказали о том, что они представляют собой изнутри, какие подходы существуют к их построению.

Монолитное ядро

По сути дела, операционная система – это обычная программа, поэтому было бы логично и ее организовать так же, как устроено большинство программ, то есть составить из процедур и функций. В этом случае компоненты операционной системы являются не самостоятельными модулями, а составными частями одной большой программы. Такая структура операционной системы называется *монолитным ядром (monolithic kernel)*. Монолитное ядро представляет собой набор процедур, каждая из которых может вызвать каждую. Все процедуры работают в привилегированном режиме. Таким образом, монолитное ядро – это такая схема операционной системы, при которой все ее компоненты являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путем непосредственного вызова процедур. Для монолитной операционной системы ядро совпадает со всей системой.

Во многих операционных системах с монолитным ядром сборка ядра, то есть его компиляция, осуществляется отдельно для каждого компьютера, на который устанавливается операционная система. При этом можно выбрать список оборудования и программных протоколов, поддержка которых будет включена в ядро. Так как ядро является единой программой, перекомпиляция – это единственный способ добавить в него новые компоненты или исключить неиспользуемые. Следует отметить, что присутствие в ядре лишних компонентов крайне нежелательно, так как ядро всегда полностью располагается в оперативной памяти. Кроме того, исключение ненужных компонентов повышает надежность операционной системы в целом.

Монолитное ядро – старейший способ организации операционных систем. Примером систем с монолитным ядром является большинство Unix-систем.

Даже в монолитных системах можно выделить некоторую структуру. Как в бетонной глыбе можно различить вкрапления щебенки, так и в монолитном ядре выделяются вкрапления сервисных процедур, соответствующих системным вызовам. Сервисные процедуры выполняются в привилегированном режиме, тогда как пользовательские программы –

в непривилегированном. Для перехода с одного уровня привилегий на другой иногда может использоваться главная сервисная программа, определяющая, какой именно системный вызов был сделан, корректность входных данных для этого вызова и передающая управление соответствующей сервисной процедуре с переходом в привилегированный режим работы. Иногда выделяют также набор программных утилит, которые помогают выполнять сервисные процедуры.

Многоуровневые системы (Layered systems)

Продолжая структуризацию, можно разбить всю вычислительную систему на ряд более мелких уровней с хорошо определенными связями между ними, так чтобы объекты уровня N могли вызывать только объекты уровня N-1. Нижним уровнем в таких системах обычно является hardware, верхним уровнем – интерфейс пользователя. Чем ниже уровень, тем более привилегированные команды и действия может выполнять модуль, находящийся на этом уровне. Впервые такой подход был применен при создании системы THE (Technishe Hogeschool Eindhoven) Дейкстрой (Dijkstra) и его студентами в 1968 г. Эта система имела следующие уровни:

5	Интерфейс пользователя
4	Управление вводом-выводом
3	Драйвер устройства связи оператора и консоли
2	Управление памятью
1	Планирование задач и процессов
0	Hardware

Рис. 1.2. Слоеная система THE

Слоеные системы хорошо реализуются. При использовании операций нижнего слоя не нужно знать, как они реализованы — нужно лишь понимать, что они делают. Слоеные системы хорошо тестируются. Отладка начинается с нижнего слоя и проводится послойно. При возникновении ошибки мы можем быть уверены, что она находится в тестируемом слое. Слоеные системы хорошо модифицируются. При необходимости можно заменить лишь один слой, не трогая остальные. Но слоеные системы сложны для разработки: тяжело правильно определить порядок слоев и что к какому слою относится. Слоеные системы менее эффективны, чем монолитные. Так, например, для выполнения операций ввода-вывода программе пользователя придется последовательно проходить все слои от верхнего до нижнего.

Виртуальные машины

В начале лекции мы говорили о взгляде на операционную систему как на виртуальную машину, когда пользователю нет необходимости знать детали внутреннего устройства компьютера. Он работает с файлами, а не с магнитными головками и двигателем; он работает с огромной виртуальной, а не ограниченной реальной оперативной памятью; его мало волнует, единственный он на машине пользователь или нет. Рассмотрим несколько иной подход. Пусть операционная система реализует виртуальную машину для каждого пользователя, но не упрощая ему жизнь, а, наоборот, усложняя. Каждая такая виртуальная машина предстает перед пользователем как голое железо – копия всего hardware в вычислительной системе, включая процессор, привилегированные и непривилегированные команды, устройства ввода-вывода, прерывания и т. д. И он остается с этим железом один на один. При попытке обратиться к такому виртуальному железу на уровне привилегированных команд в действительности, происходит системный вызов реальной операционной системы, которая и производит все необходимые действия. Такой подход позволяет каждому пользователю загрузить свою операционную систему на виртуальную машину и делать с ней все, что душа пожелает.

Программа пользователя	Программа пользователя	Программа пользователя
MS-DOS	Linux	Windows-NT
Виртуальное hardware	Виртуальное hardware	Виртуальное hardware
Реальная операционная система		
Реальное hardware		

Рис. 1.3. Вариант виртуальной машины

Первой реальной системой такого рода была система CP/CMS, или VM/370, как ее называют сейчас, для семейства машин IBM/370. Недостатком таких операционных систем является снижение эффективности виртуальных машин по сравнению с реальным компьютером, и, как правило, они очень громоздки. Преимущество же заключается в использовании на одной вычислительной системе программ, написанных для разных операционных систем.

Микроядерная архитектура

Современная тенденция в разработке операционных систем состоит в перенесении значительной части системного кода на уровень пользователя и одновременной минимизации ядра. Речь идет о подходе к построению ядра, называемом *микроядерной архитектурой* (microkernel architecture) операционной системы, когда большинство ее составляющих являются самостоятельными программами. В этом случае взаимодействие между ними обеспечивает специальный модуль ядра, называемый микроядром. Микроядро работает в привилегированном режиме и обеспечивает взаимодействие между программами, планирование использования процессора, первичную обработку прерываний, операции ввода-вывода и базовое управление памятью.

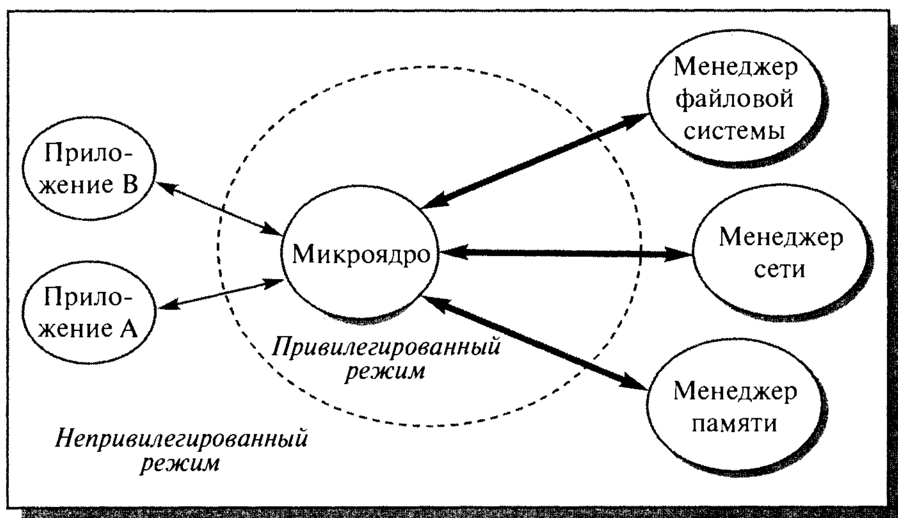


Рис. 1.4. Микроядерная архитектура операционной системы

Остальные компоненты системы взаимодействуют друг с другом путем передачи сообщений через микроядро.

Основное достоинство микроядерной архитектуры – высокая степень модульности ядра операционной системы. Это существенно упрощает добавление в него новых компонентов. В микроядерной операционной системе можно, не прерывая ее работы, загружать и выгружать новые драйверы, файловые системы и т.д. Существенно упрощается процесс отладки компонентов ядра, так как новая версия драйвера может загружаться без перезапуска всей операционной системы. Компоненты ядра операционной

системы ничем принципиально не отличаются от пользовательских программ, поэтому для их отладки можно применять обычные средства. Микроядерная архитектура повышает надежность системы, поскольку ошибка на уровне непривилегированной программы менее опасна, чем отказ на уровне режима ядра.

В то же время микроядерная архитектура операционной системы вносит дополнительные накладные расходы, связанные с передачей сообщений, что существенно влияет на производительность. Для того чтобы микроядерная операционная система по скорости не уступала операционным системам на базе монолитного ядра, требуется очень аккуратно проектировать разбиение системы на компоненты, стараясь минимизировать взаимодействие между ними. Таким образом, основная сложность при создании микроядерных операционных систем – необходимость очень аккуратного проектирования.

Смешанные системы

Все рассмотренные подходы к построению операционных систем имеют свои достоинства и недостатки. В большинстве случаев современные операционные системы используют различные комбинации этих подходов. Так, например, ядро операционной системы Linux представляет собой монолитную систему с элементами микроядерной архитектуры. При компиляции ядра можно разрешить динамическую загрузку и выгрузку очень многих компонентов ядра – так называемых модулей. В момент загрузки модуля его код загружается на уровне системы и связывается с остальной частью ядра. Внутри модуля могут использоваться любые экспортируемые ядром функции.

Другим примером смешанного подхода может служить возможность запуска операционной системы с монолитным ядром под управлением микроядра. Так устроены 4.4BSD и MkLinux, основанные на микроядре Mach. Микроядро обеспечивает управление виртуальной памятью и работу низкоуровневых драйверов. Все остальные функции, в том числе взаимодействие с прикладными программами, осуществляются монолитным ядром. Данный подход сформировался в результате попыток использования преимущества микроядерной архитектуры, сохраняя по возможности хорошо отлаженный код монолитного ядра.

Наиболее тесно элементы микроядерной архитектуры и элементы монолитного ядра переплетены в ядре Windows NT. Хотя Windows NT часто называют микроядерной операционной системой, это не совсем так. Микроядро NT слишком велико (более 1 Мбайт), чтобы носить приставку «микро». Компоненты ядра Windows NT располагаются в вытесняемой памяти и взаимодействуют друг с другом путем передачи сообщений, как

и положено в микроядерных операционных системах. В то же время все компоненты ядра работают в одном адресном пространстве и активно используют общие структуры данных, что свойственно операционным системам с монолитным ядром. По мнению специалистов Microsoft, причина проста: чисто микроядерный дизайн коммерчески невыгоден, поскольку неэффективен.

Таким образом, Windows NT можно с полным правом назвать гибридной операционной системой.

Классификация ОС

Существует несколько схем классификации операционных систем. Ниже приведена классификация по некоторым признакам с точки зрения пользователя.

Реализация многозадачности

По числу одновременно выполняемых задач операционные системы можно разделить на два класса:

- многозадачные (Unix, OS/2, Windows);
- однозадачные (например, MS-DOS).

Многозадачная ОС, решая проблемы распределения ресурсов и конкуренции, полностью реализует мультипрограммный режим в соответствии с требованиями раздела «Основные понятия, концепции ОС».

Многозадачный режим, который воплощает в себе идею разделения времени, называется вытесняющим (preemptive). Каждой программе выделяется квант процессорного времени, по истечении которого управление передается другой программе. Говорят, что первая программа будет вытеснена. В вытесняющем режиме работают пользовательские программы большинства коммерческих ОС.

В некоторых ОС (Windows 3.11, например) пользовательская программа может монополизировать процессор, то есть работает в невытесняющем режиме. Как правило, в большинстве систем не подлежит вытеснению код собственно ОС. Ответственные программы, в частности задачи реального времени, также не вытесняются. Более подробно об этом рассказано в лекции, посвященной планированию работы процессора.

По приведенным примерам можно судить о приблизительности классификации. Так, в ОС MS-DOS можно организовать запуск дочерней задачи и наличие в памяти двух и более задач одновременно. Однако эта ОС традиционно считается однозадачной, главным образом из-за отсутствия защитных механизмов и коммуникационных возможностей.

Поддержка многопользовательского режима

По числу одновременно работающих пользователей ОС можно разделить на:

- однопользовательские (MS-DOS, Windows 3.x);
- многопользовательские (Windows NT, Unix).

Наиболее существенное отличие между этими ОС заключается в наличии у многопользовательских систем механизмов защиты персональных данных каждого пользователя.

Многопроцессорная обработка

Вплоть до недавнего времени вычислительные системы имели один центральный процессор. В результате требований к повышению производительности появились многопроцессорные системы, состоящие из двух и более процессоров общего назначения, осуществляющих параллельное выполнение команд. Поддержка мультипроцессирования является важным свойством ОС и приводит к усложнению всех алгоритмов управления ресурсами. Многопроцессорная обработка реализована в таких ОС, как Linux, Solaris, Windows NT и ряде других.

Многопроцессорные ОС разделяют на симметричные и асимметричные. В симметричных ОС на каждом процессоре функционирует одно и то же ядро, и задача может быть выполнена на любом процессоре, то есть обработка полностью децентрализована. При этом каждому из процессоров доступна вся память.

В асимметричных ОС процессоры неравноправны. Обычно существует главный процессор (master) и подчиненные (slave), загрузку и характер работы которых определяет главный процессор.

Системы реального времени

В разряд многозадачных ОС, наряду с пакетными системами и системами разделения времени, включаются также *системы реального времени*, не упоминавшиеся до сих пор.

Они используются для управления различными техническими объектами или технологическими процессами. Такие системы характеризуются предельно допустимым временем реакции на внешнее событие, в течение которого должна быть выполнена программа, управляющая объектом. Система должна обрабатывать поступающие данные быстрее, чем они могут поступать, причем от нескольких источников одновременно.

Столь жесткие ограничения сказываются на архитектуре систем реального времени, например, в них может отсутствовать виртуальная

память, поддержка которой дает непредсказуемые задержки в выполнении программ. (См. также разделы, связанные с планированием процессов и реализацией виртуальной памяти.)

Приведенная классификация ОС не является исчерпывающей. Более подробно особенности применения современных ОС рассмотрены в [Олифер, 2001].

Заключение

Мы рассмотрели различные взгляды на то, что такое операционная система; изучили историю развития операционных систем; выяснили, какие функции обычно выполняют операционные системы; наконец, разобрались в том, какие существуют подходы к построению операционных систем. Следующую лекцию мы посвятим выяснению понятия «процесс» и вопросам планирования процессов.

Приложение 1.
Некоторые сведения об архитектуре компьютера

Основными аппаратными компонентами компьютера являются: основная память, центральный процессор и периферийные устройства. Для обмена данными между собой эти компоненты соединены группой проводов, называемой магистралью.

Рис. 1.5. Некоторые компоненты компьютера

Основная память используется для запоминания программ и данных в двоичном виде и организована в виде упорядоченного массива ячеек, каждая из которых имеет уникальный цифровой адрес. Как правило, размер ячейки составляет один байт. Типовые операции над основной памятью – считывание и запись содержимого ячейки с определенным адресом.

Выполнение различных операций с данными осуществляется изолированной частью компьютера, называемой центральным процессором

(ЦП). ЦП также имеет ячейки для запоминания информации, называемые регистрами. Их разделяют на регистры общего назначения и специализированные регистры. В современных компьютерах емкость регистра обычно составляет 4-8 байт. Регистры общего назначения используются для временного хранения данных и результатов операций. Для обработки информации обычно организовывается передача данных из ячеек памяти в регистры общего назначения, выполнение операции центральным процессором и передача результатов операции в основную память.

Специализированные регистры используются для контроля работы процессора. Наиболее важными являются: программный счетчик, регистр команд и регистр, содержащий информацию о состоянии программы.

Программы хранятся в виде последовательности машинных команд, которые должен выполнять центральный процессор. Каждая команда состоит из поля операции и полей операндов, то есть тех данных, над которыми выполняется данная операция. Весь набор машинных команд называется машинным языком.

Выполнение программы осуществляется следующим образом. Машинная команда, на которую указывает программный счетчик, считывается из памяти и копируется в регистр команд. Здесь она декодируется, после чего исполняется. После выполнения команды программный счетчик указывает на следующую команду. Затем эти действия, называемые машинным циклом, повторяются.

Взаимодействие с периферийными устройствами

Периферийные устройства предназначены для ввода и вывода информации. Каждое устройство обычно имеет в своем составе специализированный компьютер, называемый контроллером или адаптером. Когда контроллер вставляется в разъем на материнской плате, он подключается к шине и получает уникальный номер (адрес). После этого контроллер осуществляет наблюдение за сигналами, идущими по шине, и отвечает на сигналы, адресованные ему.

Любая операция ввода-вывода предполагает диалог между ЦП и контроллером устройства. Когда процессору встречается команда, связанная с вводом-выводом, входящая в состав какой-либо программы, он выполняет ее, посылая сигналы контроллеру устройства. Это так называемый программируемый ввод-вывод.

В свою очередь, любые изменения с внешними устройствами имеют следствием передачу сигнала от устройства к ЦП. С точки зрения ЦП это является асинхронным событием и требует его реакции. Для того чтобы обнаружить такое событие, между машинными циклами процессор опрашивает специальный регистр, содержащий информацию о типе устройства,

сгенерировавшего сигнал. Если сигнал имеет место, то ЦП выполняет специфичную для данного устройства программу, задача которой – отреагировать на это событие надлежащим образом (например, занести символ, введенный с клавиатуры, в специальный буфер). Такая программа называется программой обработки прерывания, а само событие — прерыванием, поскольку оно нарушает плановую работу процессора. После завершения обработки прерывания процессор возвращается к выполнению программы. Эти действия компьютера называются вводом-выводом с использованием прерываний.

В современных компьютерах также имеется возможность непосредственного взаимодействия между контроллером и основной памятью, минуя ЦП, – так называемый механизм прямого доступа к памяти.

Часть II. Процессы и их поддержка в операционной системе

Лекция 2. Процессы

В лекции описывается основополагающее понятие процесса, рассматриваются его состояния, модель представления процесса в операционной системе и операции, которые могут выполняться над процессами операционной системой.

Ключевые слова: процесс, состояния процесса, состояние *рождение*, состояние *готовность*, состояние *исполнение*, состояние *ожидание*, состояние *закончил исполнение*, операции над процессами, создание процесса, завершение процесса, приостановка процесса, запуск процесса, блокирование процесса, разблокирование процесса, блок управления процессом (PCB), контекст процесса, системный контекст, пользовательский контекст, регистровый контекст, переключение контекста.

Начиная с этой лекции мы будем знакомиться с внутренним устройством и механизмами действия операционных систем, разбирая одну за другой их основные функции по отдельности и во взаимосвязи. Фундаментальным понятием для изучения работы операционных систем является понятие процессов как основных динамических объектов, над которыми системы выполняют определенные действия. Данная лекция посвящена описанию таких объектов, их состояний и свойств, их представлению в вычислительных системах, а также операциям, которые могут проводиться над ними.

Понятие процесса

В первой части книги, поясняя понятие «операционная система» и описывая способы построения операционных систем, мы часто применяли слова «программа» и «задание». Мы говорили: вычислительная система исполняет одну или несколько программ, операционная система планирует задания, программы могут обмениваться данными и т. д. Мы использовали эти термины в некотором общеупотребительном, житейском смысле, предполагая, что все читатели одинаково представляют себе, что подразумевается под ними в каждом конкретном случае. При этом одни и те же слова обозначали и объекты в статическом состоянии, не обрабатывающи-

еся вычислительной системой (например, совокупность файлов на диске), и объекты в динамическом состоянии, находящиеся в процессе исполнения. Это было возможно, пока мы говорили об общих свойствах операционных систем, не вдаваясь в подробности их внутреннего устройства и поведения, или о работе вычислительных систем первого-второго поколений, которые не могли обрабатывать более одной программы или одного задания одновременно, по сути дела не имея операционных систем. Но теперь мы начинаем знакомиться с деталями функционирования современных компьютерных систем, и нам придется уточнить терминологию.

Рассмотрим следующий пример. Два студента запускают программу извлечения квадратного корня. Один хочет вычислить квадратный корень из 4, а второй – из 1. С точки зрения студентов, запущена одна и та же программа; с точки зрения компьютерной системы, ей приходится заниматься двумя различными вычислительными процессами, так как разные исходные данные приводят к разному набору вычислений. Следовательно, на уровне происходящего внутри вычислительной системы мы не можем использовать термин «программа» в пользовательском смысле слова.

Рассматривая системы пакетной обработки, мы ввели понятие «задание» как совокупность программы, набора команд языка управления заданиями, необходимых для ее выполнения, и входных данных. С точки зрения студентов, они, подставив разные исходные данные, сформировали два различных задания. Может быть, термин «задание» подойдет нам для описания внутреннего функционирования компьютерных систем? Чтобы выяснить это, давайте рассмотрим другой пример. Пусть оба студента пытаются извлечь квадратный корень из 1, то есть пусть они сформировали идентичные задания, но загрузили их в вычислительную систему со сдвигом по времени. В то время как одно из выполняемых заданий приступило к печати полученного значения и ждет окончания операции ввода-вывода, второе только начинает исполняться. Можно ли говорить об идентичности заданий внутри вычислительной системы в данный момент? Нет, так как состояние процесса их выполнения различно. Следовательно, и слово «задание» в пользовательском смысле не может применяться для описания происходящего в вычислительной системе.

Это происходит потому, что термины «программа» и «задание» предназначены для описания статических, неактивных объектов. Программа же в процессе исполнения является динамическим, активным объектом. По ходу ее работы компьютер обрабатывает различные команды и преобразует значения переменных. Для выполнения программы операционная система должна выделить определенное количество оперативной памяти, закрепить за ней определенные устройства ввода-вывода или файлы (откуда должны поступать входные данные и куда нужно доставить полученные результаты), то есть зарезервировать определенные ресурсы из общего чис-

ла ресурсов всей вычислительной системы. Их количество и конфигурация с течением времени могут изменяться. Для описания таких активных объектов внутри компьютерной системы вместо терминов «программа» и «задание» мы будем использовать новый термин – «процесс».

В ряде учебных пособий и монографий для простоты предлагается рассматривать процесс как абстракцию, характеризующую программу во время выполнения. На наш взгляд, эта рекомендация не совсем корректна. Понятие процесса характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или *адресное пространство*, стеки, используемые файлы и устройства ввода-вывода, и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы. Не существует взаимно-однозначного соответствия между процессами и программами, обрабатываемыми вычислительными системами. Как будет показано далее, в некоторых операционных системах для работы определенных программ может организовываться более одного процесса или один и тот же процесс может исполнять последовательно несколько различных программ. Более того, даже в случае обработки только одной программы в рамках одного процесса нельзя считать, что процесс представляет собой просто динамическое описание кода исполняемого файла, данных и выделенных для них ресурсов. Процесс находится под управлением операционной системы, поэтому в нем может выполняться часть кода ее ядра (не находящегося в исполняемом файле!), как в случаях, специально запланированных авторами программы (например, при использовании системных вызовов), так и в непредусмотренных ситуациях (например, при обработке внешних прерываний).

Состояния процесса

При использовании такой абстракции все, что выполняется в вычислительных системах (не только программы пользователей, но и, возможно, определенные части операционных систем), организовано как набор процессов. Понятно, что реально на однопроцессорной компьютерной системе в каждый момент времени может исполняться только один процесс. Для мультипрограммных вычислительных систем псевдопараллельная обработка нескольких процессов достигается с помощью переключения процессора с одного процесса на другой. Пока один процесс выполняется, остальные ждут своей очереди.

Как видим, каждый процесс может находиться как минимум в двух состояниях: ***процесс исполняется*** и ***процесс не исполняется***. Диаграмма состояний процесса в такой модели изображена на рис. 2.1.

Рис. 2.1. Простейшая диаграмма состояний процесса

Процесс, находящийся в состоянии ***процесс исполняется***, через некоторое время может быть завершен операционной системой или приостановлен и снова переведен в состояние ***процесс не исполняется***. Приостановка процесса происходит по двум причинам: для его дальнейшей работы потребовалось какое-либо событие (например, завершение операции ввода-вывода) или истек временной интервал, отведенный операционной системой для работы данного процесса. После этого операционная система по определенному алгоритму выбирает для исполнения один из процессов, находящихся в состоянии ***процесс не исполняется***, и переводит его в состояние ***процесс исполняется***. Новый процесс, появляющийся в системе, первоначально помещается в состояние ***процесс не исполняется***.

Это очень грубая модель, она не учитывает, в частности, то, что процесс, выбранный для исполнения, может все еще ждать события, из-за которого он был приостановлен, и реально к выполнению не готов. Для того чтобы избежать такой ситуации, разобьем состояние ***процесс не исполняется*** на два новых состояния: ***готовность*** и ***ожидание*** (см. рис. 2.2).

Всякий новый процесс, появляющийся в системе, попадает в состояние ***готовность***. Операционная система, пользуясь каким-либо алгоритмом планирования, выбирает один из готовых процессов и переводит его в состояние ***исполнение***. В состоянии ***исполнение*** происходит непосредственное выполнение программного кода процесса. Выйти из этого состояния процесс может по трем причинам:

- операционная система прекращает его деятельность;
- он не может продолжать свою работу, пока не произойдет некоторое событие, и операционная система переводит его в состояние ***ожидание***;
- в результате возникновения прерывания в вычислительной системе (например, прерывания от таймера по истечении предусмотренного времени выполнения) его возвращают в состояние ***готовность***.

Из состояния ***ожидание*** процесс попадает в состояние ***готовность*** после того, как ожидаемое событие произошло, и он снова может быть выбран для исполнения.

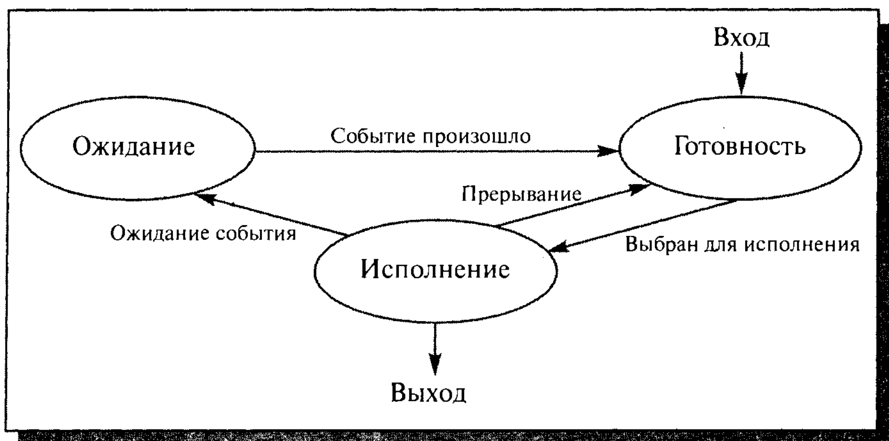


Рис. 2.2. Более подробная диаграмма состояний процесса

Наша новая модель хорошо описывает поведение процессов во время их существования, но она не акцентирует внимания на появлении процесса в системе и его исчезновении. Для полноты картины нам необходимо ввести еще два состояния процессов: ***рождение*** и ***закончил исполнение*** (см. рис. 2.3).

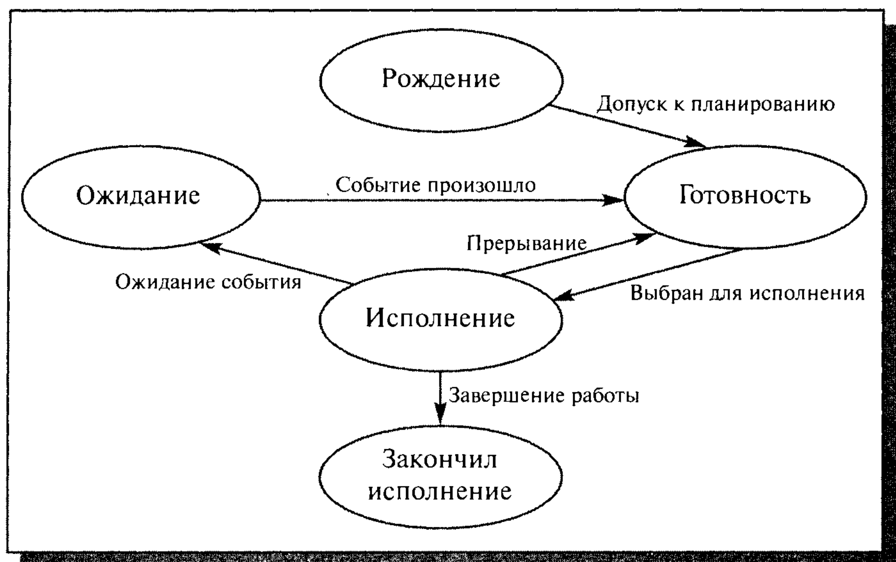


Рис. 2.3. Диаграмма состояний процесса, принятая в курсе

Теперь для появления в вычислительной системе процесс должен пройти через состояние ***рождение***. При рождении процесс получает в свое распоряжение адресное пространство, в которое загружается программный код процесса; ему выделяются стек и системные ресурсы; устанавливается начальное значение программного счетчика этого процесса и т. д. Родившийся процесс переводится в состояние ***готовность***. При завершении своей деятельности процесс из состояния ***исполнение*** попадает в состояние ***закончил исполнение***.

В конкретных операционных системах состояния процесса могут быть еще более детализированы, могут появиться некоторые новые варианты переходов из одного состояния в другое. Так, например, модель состояний процессов для операционной системы Windows NT содержит 7 различных состояний, а для операционной системы Unix – 9. Тем не менее, все операционные системы подчиняются изложенной выше модели.

Операции над процессами и связанные с ними понятия

Набор операций

Процесс не может перейти из одного состояния в другое самостоятельно. Изменением состояния процессов занимается операционная система, совершая операции над ними. Количество таких операций в нашей модели пока совпадает с количеством стрелок на диаграмме состояний. Удобно объединить их в три пары:

- создание процесса – завершение процесса;
- приостановка процесса (перевод из состояния ***исполнение*** в состояние ***готовность***) – запуск процесса (перевод из состояния ***готовность*** в состояние ***исполнение***);
- блокирование процесса (перевод из состояния ***исполнение*** в состояние ***ожидание***) – разблокирование процесса (перевод из состояния ***ожидание*** в состояние ***готовность***).

В дальнейшем, когда мы будем говорить об алгоритмах планирования, в нашей модели появится еще одна операция, не имеющая парной: изменение приоритета процесса.

Операции создания и завершения процесса являются одноразовыми, так как применяются к процессу не более одного раза (некоторые системные процессы при работе вычислительной системы не завершаются никогда). Все остальные операции, связанные с изменением состояния процессов, будь то запуск или блокировка, как правило, являются много-

разовыми. Рассмотрим подробнее, как операционная система выполняет операции над процессами.

Process Control Block и контекст процесса

Для того чтобы операционная система могла выполнять операции над процессами, каждый процесс представляется в ней некоторой структурой данных. Эта структура содержит информацию, специфическую для данного процесса:

- состояние, в котором находится процесс;
- программный счетчик процесса или, другими словами, адрес команды, которая должна быть выполнена для него следующей;
- содержимое регистров процессора;
- данные, необходимые для планирования использования процессора и управления памятью (приоритет процесса, размер и расположение адресного пространства и т. д.);
- учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом и т. д.);
- сведения об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены за процессом, таблицу открытых файлов).

Ее состав и строение зависят, конечно, от конкретной операционной системы. Во многих операционных системах информация, характеризующая процесс, хранится не в одной, а в нескольких связанных структурах данных. Эти структуры могут иметь различные наименования, содержать дополнительную информацию или, наоборот, лишь часть описанной информации. Для нас это не имеет значения. Для нас важно лишь то, что для любого процесса, находящегося в вычислительной системе, вся информация, необходимая для совершения операций над ним, доступна операционной системе. Для простоты изложения будем считать, что она хранится в одной структуре данных. Мы будем называть ее PCB (Process Control Block) или блоком управления процессом. Блок управления процессом является моделью процесса для операционной системы. Любая операция, производимая операционной системой над процессом, вызывает определенные изменения в PCB. В рамках принятой модели состояний процессов содержимое PCB между операциями остается постоянным.

Информацию, для хранения которой предназначен блок управления процессом, удобно для дальнейшего изложения разделить на две части. Содержимое всех регистров процессора (включая значение программного счетчика) будем называть *регистровым контекстом* процесса, а все остальное – *системным контекстом* процесса. Знания регистрового и системного

контекстов процесса достаточно для того, чтобы управлять его работой в операционной системе, совершая над ним операции. Однако этого недостаточно для того, чтобы полностью охарактеризовать процесс. Операционную систему не интересует, какими именно вычислениями занимается процесс, т. е. какой код и какие данные находятся в его адресном пространстве. С точки зрения пользователя, наоборот, наибольший интерес представляет содержимое адресного пространства процесса, возможно, наряду с регистровым контекстом определяющее последовательность преобразования данных и полученные результаты. Код и данные, находящиеся в адресном пространстве процесса, будем называть его *пользовательским контекстом*. Совокупность регистрового, системного и пользовательского контекстов процесса для краткости принято называть просто *контекстом* процесса. В любой момент времени процесс полностью характеризуется своим контекстом.

Одноразовые операции

Сложный жизненный путь процесса в компьютере начинается с его рождения. Любая операционная система, поддерживающая концепцию процессов, должна обладать средствами для их создания. В очень простых системах (например, в системах, спроектированных для работы только одного конкретного приложения) все процессы могут быть порождены на этапе старта системы. Более сложные операционные системы создают процессы динамически, по мере необходимости. Инициатором рождения нового процесса после старта операционной системы может выступить либо процесс пользователя, совершивший специальный системный вызов, либо сама операционная система, то есть, в конечном итоге, тоже некоторый процесс. Процесс, инициировавший создание нового процесса, принято называть *процессом-родителем (parent process)*, а вновь созданный процесс – *процессом-ребенком (child process)*. Процессы-дети могут в свою очередь порождать новых детей и т. д., образуя, в общем случае, внутри системы набор генеалогических деревьев процессов – генеалогический лес. Пример генеалогического леса изображен на рисунке 2.4. Следует отметить, что все пользовательские процессы вместе с некоторыми процессами операционной системы принадлежат одному и тому же дереву леса. В ряде вычислительных систем лес вообще вырождается в одно такое дерево.

При рождении процесса система заводит новый PCB с состоянием процесса ***рождение*** и начинает его заполнять. Новый процесс получает собственный уникальный идентификационный номер. Поскольку для хранения идентификационного номера процесса в операционной системе отводится ограниченное количество битов, для соблюдения уникальности

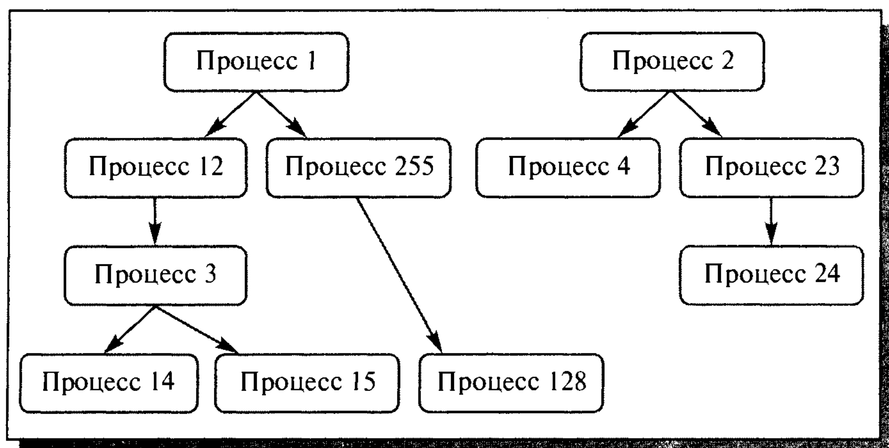


Рис. 2.4. Упрощенный генеалогический лес процессов. Стрелочка означает отношение родитель–ребенок

номеров количество одновременно присутствующих в ней процессов должно быть ограничено. После завершения какого-либо процесса его освободившийся идентификационный номер может быть повторно использован для другого процесса.

Обычно для выполнения своих функций процесс-ребенок требует определенных ресурсов: памяти, файлов, устройств ввода-вывода и т. д. Существует два подхода к их выделению. Новый процесс может получить в свое распоряжение некоторую часть родительских ресурсов, возможно, разделяя с процессом-родителем и другими процессами-детьми права на них, или может получить свои ресурсы непосредственно от операционной системы. Информация о выделенных ресурсах заносится в PCB.

После наделения процесса-ребенка ресурсами необходимо занести в его адресное пространство программный код, значения данных, установить программный счетчик. Здесь также возможны два решения. В первом случае процесс-ребенок становится дубликатом процесса-родителя по регистровому и пользовательскому контекстам, при этом должен существовать способ определения, кто для кого из процессов-двойников является родителем. Во втором случае процесс-ребенок загружается новой программой из какого-либо файла. Операционная система Unix разрешает порождение процесса только первым способом; для запуска новой программы необходимо сначала создать копию процесса-родителя, а затем процесс-ребенок должен заменить свой пользовательский контекст с помощью специального системного вызова. Операционная система VAX/VMS допускает только второе решение. В Windows NT возможны оба варианта (в различных API).

Порождение нового процесса как дубликата процесса-родителя приводит к возможности существования программ (т. е. исполняемых файлов), для работы которых организуется более одного процесса. Возможность замены пользовательского контекста процесса по ходу его работы (т. е. загрузки для исполнения новой программы) приводит к тому, что в рамках одного и того же процесса может последовательно выполняться несколько различных программ.

После того как процесс наделен содержанием, в PCB дописывается оставшаяся информация, и состояние нового процесса изменяется на ***готовность***. Осталось сказать несколько слов о том, как ведут себя процессы-родители после рождения процессов-детей. Процесс-родитель может продолжать свое выполнение одновременно с выполнением процесса-ребенка, а может ожидать завершения работы некоторых или всех своих «детей».

Мы не будем подробно останавливаться на причинах, которые могут привести к завершению жизненного цикла процесса. После того как процесс завершил свою работу, операционная система переводит его в состояние ***закончил исполнение*** и освобождает все ассоциированные с ним ресурсы, делая соответствующие записи в блоке управления процессом. При этом сам PCB не уничтожается, а остается в системе еще некоторое время. Это связано с тем, что процесс-родитель после завершения процесса-ребенка может запросить операционную систему о причине «смерти» порожденного им процесса и/или статистическую информацию о его работе. Подобная информация сохраняется в PCB отработавшего процесса до запроса процесса-родителя или до конца его деятельности, после чего все следы завершившегося процесса окончательно исчезают из системы. В операционной системе Unix процессы, находящиеся в состоянии ***закончил исполнение***, принято называть процессами-зомби.

Следует заметить, что в ряде операционных систем (например, в VAX/VMS) гибель процесса-родителя приводит к завершению работы всех его «детей». В других операционных системах (например, в Unix) процессы-дети продолжают свое существование и после окончания работы процесса-родителя. При этом возникает необходимость изменения информации в PCB процессов-детей о породившем их процессе для того, чтобы генеалогический лес процессов оставался целостным. Рассмотрим следующий пример. Пусть процесс с номером 2515 был порожден процессом с номером 2001 и после завершения его работы остается в вычислительной системе неограниченно долго. Тогда не исключено, что номер 2001 будет использован операционной системой повторно для совсем другого процесса. Если не изменить информацию о процессе-родителе для процесса 2515, то генеалогический лес процессов окажется некорректным – процесс 2515 будет считать своим родителем новый процесс

2001, а процесс 2001 будет открещиваться от нежданного потомка. Как правило, «осиротевшие» процессы «усыновляются» одним из системных процессов, который порождается при старте операционной системы и функционирует все время, пока она работает.

Многоразовые операции

Одноразовые операции приводят к изменению количества процессов, находящихся под управлением операционной системы, и всегда связаны с выделением или освобождением определенных ресурсов. Многоразовые операции, напротив, не приводят к изменению количества процессов в операционной системе и не обязаны быть связанными с выделением или освобождением ресурсов.

В этом разделе мы кратко опишем действия, которые производит операционная система при выполнении многоразовых операций над процессами. Более подробно эти действия будут рассмотрены далее в соответствующих лекциях.

Запуск процесса. Из числа процессов, находящихся в состоянии ***готовность***, операционная система выбирает один процесс для последующего исполнения. Критерии и алгоритмы такого выбора будут подробно рассмотрены в лекции 3 – «Планирование процессов». Для избранного процесса операционная система обеспечивает наличие в оперативной памяти информации, необходимой для его дальнейшего выполнения. То, как она это делает, будет в деталях описано в части III – «Управление памятью». Далее состояние процесса изменяется на ***исполнение***, восстанавливаются значения регистров для данного процесса и управление передается команде, на которую указывает счетчик команд процесса. Все данные, необходимые для восстановления контекста, извлекаются из PCB процесса, над которым совершается операция.

Приостановка процесса. Работа процесса, находящегося в состоянии ***исполнение***, приостанавливается в результате какого-либо прерывания. Процессор автоматически сохраняет счетчик команд и, возможно, один или несколько регистров в стеке исполняемого процесса, а затем передает управление по специальному адресу обработки данного прерывания. На этом деятельность hardware по обработке прерывания завершается. По указанному адресу обычно располагается одна из частей операционной системы. Она сохраняет динамическую часть системного и регистрового контекстов процесса в его PCB, переводит процесс в состояние ***готовность*** и приступает к обработке прерывания, то есть к выполнению определенных действий, связанных с возникшим прерыванием.

Блокирование процесса. Процесс блокируется, когда он не может продолжать работу, не дождавшись возникновения какого-либо события

в вычислительной системе. Для этого он обращается к операционной системе с помощью определенного системного вызова. Операционная система обрабатывает системный вызов (инициализирует операцию ввода-вывода, добавляет процесс в очередь процессов, дожидающихся освобождения устройства или возникновения события, и т. д.) и, при необходимости сохранив нужную часть контекста процесса в его PCB, переводит процесс из состояния ***исполнение*** в состояние ***ожидание***. Подробнее эта операция будет рассматриваться в части V — «Управление вводом-выводом».

Разблокирование процесса. После возникновения в системе какого-либо события операционной системе нужно точно определить, какое именно событие произошло. Затем операционная система проверяет, находился ли некоторый процесс в состоянии ***ожидание*** для данного события, и если находился, переводит его в состояние ***готовность***, выполняя необходимые действия, связанные с наступлением события (инициализация операции ввода-вывода для очередного ожидающего процесса и т. п.). Эта операция, как и операция блокирования, будет подробно описана в части V — «Управление вводом-выводом».

Переключение контекста

До сих пор мы рассматривали операции над процессами изолированно, независимо друг от друга. В действительности же деятельность мультипрограммной операционной системы состоит из цепочек операций, выполняемых над различными процессами, и сопровождается переключением процессора с одного процесса на другой.

Давайте для примера упрощенно рассмотрим, как в реальности может протекать операция разблокирования процесса, ожидающего ввода-вывода (см. рис. 2.5). При исполнении процессором некоторого процесса (на рисунке – процесс 1) возникает прерывание от устройства ввода-вывода, сигнализирующее об окончании операций на устройстве. Над выполняющимся процессом производится операция приостановки. Далее операционная система разблокирует процесс, инициировавший запрос на ввод-вывод (на рисунке – процесс 2) и осуществляет запуск приостановленного или нового процесса, выбранного при выполнении планирования (на рисунке был выбран разблокированный процесс). Как мы видим, в результате обработки информации об окончании операции ввода-вывода возможна смена процесса, находящегося в состоянии ***исполнение***.

Для корректного переключения процессора с одного процесса на другой необходимо сохранить контекст исполнявшегося процесса и восстановить контекст процесса, на который будет переключен процессор. Такая процедура сохранения/восстановления работоспособности процессов называется ***переключением контекста***. Время, затраченное на

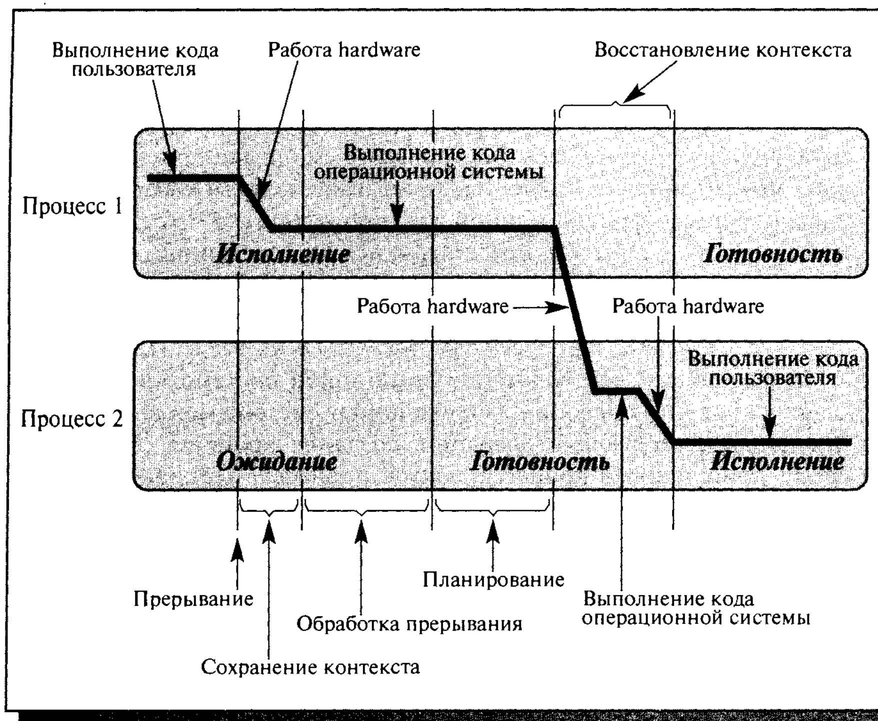


Рис. 2.5. Выполнение операции разблокирования процесса. Использование термина «код пользователя» не ограничивает общности рисунка только пользовательскими процессами

переключение контекста, не используется вычислительной системой дл: совершения полезной работы и представляет собой накладные расходы снижающие производительность системы. Оно меняется от машины : машине и обычно колеблется в диапазоне от 1 до 1000 микросекунд. Су щественно сократить накладные расходы в современных операционны системах позволяет расширенная модель процессов, включающая в себ: понятие *threads of execution* (нити исполнения или просто нити). Подроб нее о нитях исполнения мы будем говорить в лекции 4 – «Кооперация процессов и основные аспекты ее логической организации».

Заключение

Понятие процесса характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов и текущего момента его выполнения, находящуюся под управлением операционной системы. В любой момент процесс полностью описывается своим контекстом, состоящим из регистровой, системной и пользовательской частей. В операционной системе процессы представляются определенной структурой данных – PCB, отражающей содержание регистрового и системного контекстов. Процессы могут находиться в пяти основных состояниях: ***рождение***, ***готовность***, ***исполнение***, ***ожидание***, ***закончил исполнение***. Из состояния в состояние процесс переводится операционной системой в результате выполнения над ним операций. Операционная система может выполнять над процессами следующие операции: создание процесса, завершение процесса, приостановка процесса, запуск процесса, блокирование процесса, разблокирование процесса, изменение приоритета процесса. Между операциями содержимое PCB не изменяется. Деятельность мультипрограммной операционной системы состоит из цепочек перечисленных операций, выполняемых над различными процессами, и сопровождается процедурами сохранения/восстановления работоспособности процессов, т. е. переключением контекста. Переключение контекста не имеет отношения к полезной работе, выполняемой процессами, и время, затраченное на него, сокращает полезное время работы процессора.

Лекция 3. Планирование процессов

В этой лекции рассматриваются вопросы, связанные с различными уровнями планирования процессов в операционных системах. Описываются основные цели и критерии планирования, а также параметры, на которых оно основывается. Приведены различные алгоритмы планирования.

Ключевые слова: уровни планирования процессов, краткосрочное, среднесрочное и долгосрочное планирование процессов, степень мультипрограммирования, критерии планирования, turnaround time, waiting time, параметры планирования процессов, CPU burst, I/O burst, вытесняющее планирование, невытесняющее планирование, квант времени, алгоритм FCFS, алгоритм RR, алгоритм SJF, приоритет процесса, гарантированное планирование, планирование по приоритетам, многоуровневые очереди, многоуровневые очереди с обратной связью.

Я планов наших люблю громадьё...
В.В. Маяковский
Чем тщательнее мы планируем свою деятельность, тем меньше времени остается на ее осуществление.
Из анналов Госплана

Всякий раз, когда нам приходится иметь дело с ограниченным количеством ресурсов и несколькими их потребителями, будь то фонд заработной платы в трудовом коллективе или студенческая вечеринка с несколькими ящиками пива, мы вынуждены заниматься распределением наличных ресурсов между потребителями или, другими словами, планированием использования ресурсов. Такое планирование должно иметь четко поставленные цели (чего мы хотим добиться за счет распределения ресурсов) и алгоритмы, соответствующие целям и опирающиеся на параметры потребителей. Только при правильном выборе критериев и алгоритмов можно избежать таких вопросов, как: «Почему я получаю в десять раз меньше, чем мой шеф?» или «А где мое пиво?» Настоящая лекция посвящена планированию исполнения процессов в мультипрограммных вычислительных системах или, иначе говоря, планированию процессов.

Уровни планирования

В первой лекции, рассматривая эволюцию компьютерных систем, мы говорили о двух видах планирования в вычислительных системах: планировании заданий и планировании использования процессора. Пла-

нирование заданий появилось в пакетных системах после того, как для хранения сформированных пакетов заданий начали использоваться магнитные диски. Магнитные диски, являясь устройствами прямого доступа, позволяют загружать задания в компьютер в произвольном порядке, а не только в том, в котором они были записаны на диск. Изменяя порядок загрузки заданий в вычислительную систему, можно повысить эффективность ее использования. Процедуру выбора очередного задания для загрузки в машину, т. е. для порождения соответствующего процесса, мы и назвали планированием заданий. Планирование использования процессора впервые возникает в мультипрограммных вычислительных системах, где в состоянии ***готовность*** могут одновременно находиться несколько процессов. Именно для процедуры выбора из них одного процесса, который получит процессор в свое распоряжение, т. е. будет переведен в состояние ***исполнение***, мы использовали это словосочетание. Теперь, познакомившись с концепцией процессов в вычислительных системах, оба вида планирования мы будем рассматривать как различные уровни планирования процессов.

Планирование заданий используется в качестве долгосрочного планирования процессов. Оно отвечает за порождение новых процессов в системе, определяя ее *степень мультипрограммирования*, т. е. количество процессов, одновременно находящихся в ней. Если степень мультипрограммирования системы поддерживается постоянной, т. е. среднее количество процессов в компьютере не меняется, то новые процессы могут появляться только после завершения ранее загруженных. Поэтому долгосрочное планирование осуществляется достаточно редко, между появлением новых процессов могут проходить минуты и даже десятки минут. Решение о выборе для запуска того или иного процесса оказывает влияние на функционирование вычислительной системы на протяжении достаточно длительного времени. Отсюда и название этого уровня планирования – долгосрочное. В некоторых операционных системах долгосрочное планирование сведено к минимуму или отсутствует вовсе. Так, например, во многих интерактивных системах разделения времени порождение процесса происходит сразу после появления соответствующего запроса. Поддержание разумной степени мультипрограммирования осуществляется за счет ограничения количества пользователей, которые могут работать в системе, и особенностей человеческого восприятия. Если между нажатием на клавишу и появлением символа на экране проходит 20-30 секунд, то многие пользователи предпочтут прекратить работу и продолжить ее, когда система будет менее загружена.

Планирование использования процессора применяется в качестве краткосрочного планирования процессов. Оно проводится, к примеру, при обращении исполняющегося процесса к устройствам ввода-вывода

или просто по завершении определенного интервала времени. Поэтому краткосрочное планирование осуществляется, как правило, не реже одного раза в 100 миллисекунд. Выбор нового процесса для исполнения оказывает влияние на функционирование системы до наступления очередного аналогичного события, т. е. в течение короткого промежутка времени, чем и обусловлено название этого уровня планирования – краткосрочное.

В некоторых вычислительных системах бывает выгодно для повышения производительности временно удалить какой-либо частично выполнившийся процесс из оперативной памяти на диск, а позже вернуть его обратно для дальнейшего выполнения. Такая процедура в англоязычной литературе получила название *swapping*, что можно перевести на русский язык как «перекачка», хотя в специальной литературе оно употребляется без перевода – свопинг. Когда и какой из процессов нужно перекачать на диск и вернуть обратно, решается дополнительным промежуточным уровнем планирования процессов – среднесрочным.

Критерии планирования и требования к алгоритмам

Для каждого уровня планирования процессов можно предложить много различных алгоритмов. Выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой, и целями, которых мы хотим достичь, используя планирование. К числу таких целей можно отнести следующие:

- *Справедливость* – гарантировать каждому заданию или процессу определенную часть времени использования процессора в компьютерной системе, стараясь не допустить возникновения ситуации, когда процесс одного пользователя постоянно занимает процессор, в то время как процесс другого пользователя фактически не начинал выполняться.
- *Эффективность* – постараться занять процессор на все 100% рабочего времени, не позволяя ему простаивать в ожидании процессов, готовых к исполнению. В реальных вычислительных системах загрузка процессора колеблется от 40 до 90%.
- *Сокращение полного времени выполнения (turnaround time)* – обеспечить минимальное время между стартом процесса или постановкой задания в очередь для загрузки и его завершением.
- *Сокращение времени ожидания (waiting time)* – сократить время, которое проводят процессы в состоянии ***готовность*** и задания в очереди для загрузки.
- *Сокращение времени отклика (response time)* – минимизировать время, которое требуется процессу в интерактивных системах для ответа на запрос пользователя.

Независимо от поставленных целей планирования желательно также, чтобы алгоритмы обладали следующими свойствами:

- Были предсказуемыми. Одно и то же задание должно выполняться приблизительно за одно и то же время. Применение алгоритма планирования не должно приводить, к примеру, к извлечению квадратного корня из 4 за сотые доли секунды при одном запуске и за несколько суток – при втором запуске.
- Были связаны с минимальными накладными расходами. Если на каждые 100 миллисекунд, выделенные процессу для использования процессора, будет приходиться 200 миллисекунд на определение того, какой именно процесс получит процессор в свое распоряжение, и на переключение контекста, то такой алгоритм, очевидно, применять не стоит.
- Равномерно загружали ресурсы вычислительной системы, отдавая предпочтение тем процессам, которые будут занимать малоиспользуемые ресурсы.
- Обладали масштабируемостью, т. е. не сразу теряли работоспособность при увеличении нагрузки. Например, рост количества процессов в системе в два раза не должен приводить к увеличению полного времени выполнения процессов на порядок.

Многие из приведенных выше целей и свойств являются противоречивыми. Улучшая работу алгоритма с точки зрения одного критерия, мы ухудшаем ее с точки зрения другого. Приспосабливая алгоритм под один класс задач, мы тем самым дискриминируем задачи другого класса. «В одну телегу впрячь не можно коня и трепетную лань». Ничего не поделаешь. Такова жизнь.

Параметры планирования

Для осуществления поставленных целей разумные алгоритмы планирования должны опираться на какие-либо характеристики процессов в системе, заданий в очереди на загрузку, состояния самой вычислительной системы — иными словами, на параметры планирования. В этом разделе мы опишем ряд таких параметров, не претендуя на полноту изложения.

Все параметры планирования можно разбить на две большие группы: статические параметры и динамические параметры. Статические параметры не изменяются в ходе функционирования вычислительной системы, динамические же, напротив, подвержены постоянным изменениям.

К статическим параметрам вычислительной системы можно отнести предельные значения ее ресурсов (размер оперативной памяти, максимальное количество памяти на диске для осуществления свопинга, количество подключенных устройств ввода-вывода и т. п.). Динамические

параметры системы описывают количество свободных ресурсов на данный момент.

К статическим параметрам процессов относятся характеристики, как правило присущие заданиям уже на этапе загрузки:

- Каким пользователем запущен процесс или сформировано задание.
- Насколько важной является поставленная задача, т. е. каков приоритет ее выполнения.
- Сколько процессорного времени запрошено пользователем для решения задачи.
- Каково соотношение процессорного времени и времени, необходимого для осуществления операций ввода-вывода.
- Какие ресурсы вычислительной системы (оперативная память, устройства ввода-вывода, специальные библиотеки и системные программы и т. д.) и в каком количестве необходимы заданию.

Алгоритмы долгосрочного планирования используют в своей работе статические и динамические параметры вычислительной системы и статические параметры процессов (динамические параметры процессов на этапе загрузки заданий еще не известны). Алгоритмы краткосрочного и среднесрочного планирования дополнительно учитывают и динамические характеристики процессов. Для среднесрочного планирования в качестве таких характеристик может использоваться следующая информация:

- сколько времени прошло с момента выгрузки процесса на диск или его загрузки в оперативную память;
- сколько оперативной памяти занимает процесс;
- сколько процессорного времени уже предоставлено процессу.

Для краткосрочного планирования нам понадобится ввести еще два динамических параметра. Деятельность любого процесса можно представить как последовательность циклов использования процессора и ожидания завершения операций ввода-вывода. Промежуток времени непрерывного использования процессора носит название CPU burst, а промежуток времени непрерывного ожидания ввода-вывода – I/O burst. На рисунке 3.1. показан фрагмент деятельности некоторого процесса на псевдоязыке программирования с выделением указанных промежутков. Для краткости мы будем использовать термины CPU burst и I/O burst без перевода. Значения продолжительности последних и очередных CPU burst и I/O burst являются важными динамическими параметрами процесса.

Вытесняющее и невытесняющее планирование

Процесс планирования осуществляется частью операционной системы, называемой планировщиком. Планировщик может принимать ре-

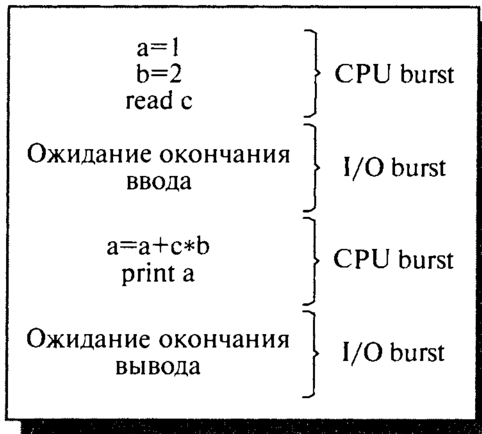


Рис. 3.1. Фрагмент деятельности процесса с выделением промежутков непрерывного использования процессора и ожидания ввода-вывода

шения о выборе для исполнения нового процесса из числа находящихся в состоянии ***готовность*** в следующих четырех случаях:

1. Когда процесс переводится из состояния ***исполнение*** в состояние ***закончил исполнение***.
2. Когда процесс переводится из состояния ***исполнение*** в состояние ***ожидание***.
3. Когда процесс переводится из состояния ***исполнение*** в состояние ***готовность*** (например, после прерывания от таймера).
4. Когда процесс переводится из состояния ***ожидание*** в состояние ***готовность*** (завершилась операция ввода-вывода или произошло другое событие). Подробно процедура такого перевода рассматривалась в лекции 2 (раздел «Переключение контекста»), где мы показали, почему при этом возникает возможность смены процесса, находящегося в состоянии ***исполнение***.

В случаях 1 и 2 процесс, находившийся в состоянии ***исполнение***, не может дальше исполняться, и операционная система вынуждена осуществлять планирование, выбирая новый процесс для выполнения. В случаях 3 и 4 планирование может как проводиться, так и не проводиться, планировщик не вынужден обязательно принимать решение о выборе процесса для выполнения, процесс, находившийся в состоянии ***исполнение*** может просто продолжить свою работу. Если в операционной системе планирование осуществляется только в вынужденных ситуациях, говорят, что имеет место *невытесняющее (nonpreemptive) планирование*. Если планировщик принимает и вынужденные, и невынужденные решения, говорят о *вытесняющем (preemptive) планировании*. Термин «вытесняющее планиро-

вание» возник потому, что исполняющийся процесс помимо своей воли может быть вытеснен из состояния ***исполнение*** другим процессом.

Невытесняющее планирование используется, например, в MS Windows 3.1 и ОС Apple Macintosh. При таком режиме планирования процесс занимает столько процессорного времени, сколько ему необходимо. При этом переключение процессов возникает только при желании самого исполняющегося процесса передать управление (для ожидания завершения операции ввода-вывода или по окончании работы). Этот метод планирования относительно просто реализуем и достаточно эффективен, так как позволяет выделить большую часть процессорного времени для работы самих процессов и до минимума сократить затраты на переключение контекста. Однако при невытесняющем планировании возникает проблема возможности полного захвата процессора одним процессом, который вследствие каких-либо причин (например, из-за ошибки в программе) зацикливается и не может передать управление другому процессу. В такой ситуации спасает только перезагрузка всей вычислительной системы.

Вытесняющее планирование обычно используется в системах разделения времени. В этом режиме планирования процесс может быть приостановлен в любой момент исполнения. Операционная система устанавливает специальный таймер для генерации сигнала прерывания по истечении некоторого интервала времени – *кванта*. После прерывания процессор передается в распоряжение следующего процесса. Временные прерывания помогают гарантировать приемлемое время отклика процессов для пользователей, работающих в диалоговом режиме, и предотвращают «зависание» компьютерной системы из-за зацикливания какой-либо программы.

Алгоритмы планирования

Существует достаточно большой набор разнообразных алгоритмов планирования, которые предназначены для достижения различных целей и эффективны для разных классов задач. Многие из них могут использоваться на нескольких уровнях планирования. В этом разделе мы рассмотрим некоторые наиболее употребительные алгоритмы применительно к процессу кратковременного планирования.

First-Come, First-Served (FCFS)

Простейшим алгоритмом планирования является алгоритм, который принято обозначать аббревиатурой FCFS по первым буквам его английского названия – First-Come, First-Served (первым пришел, первым обслужен). Представим себе, что процессы, находящиеся в состоянии

готовность, выстроены в очередь. Когда процесс переходит в состояние ***готовность***, он, а точнее, ссылка на его PCB помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на его PCB. Очередь подобного типа имеет в программировании специальное наименование – FIFO, сокращение от First In, First Out (первым вошел, первым вышел)*.

Такой алгоритм выбора процесса осуществляет невытесняющее планирование. Процесс, получивший в свое распоряжение процессор, занимает его до истечения текущего CPU burst. После этого для выполнения выбирается новый процесс из начала очереди.

Преимуществом алгоритма FCFS является легкость его реализации, но в то же время он имеет и много недостатков. Рассмотрим следующий пример. Пусть в состоянии ***готовность*** находятся три процесса — p_0, p_1 и p_2, для которых известны времена их очередных CPU burst. Эти времена приведены в таблице 3.1 в некоторых условных единицах. Для простоты будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода и что время переключения контекста так мало, что им можно пренебречь.

Таблица 3.1

Процесс	p_0	p_1	p_2
Продолжительность очередного CPU burst	13	4	1

Если процессы расположены в очереди процессов, готовых к исполнению, в порядке p_0, p_1, p_2, то картина их выполнения выглядит так, как показано на рисунке 3.2. Первым для выполнения выбирается процесс p_0, который получает процессор на все время своего CPU burst, т. е. на 13 единиц времени. После его окончания в состояние ***исполнение*** переводится процесс p_1, он занимает процессор на 4 единицы времени. И, наконец, возможность работать получает процесс p_2. Время ожидания для процесса p_0 составляет 0 единиц времени, для процесса p_1 – 13 единиц, для процесса p_2 – 13 + 4 = 17 единиц. Таким образом, среднее время ожидания в этом случае – (0 + 13 + 17)/3 = 10 единиц времени. Полное время выполнения для процесса p_0 составляет 13 единиц времени, для процесса p_1 – 13 + 4 = 17 единиц, для процесса p_2 – 13 + 4 + 1 = 18 единиц. Среднее пол-

* Надо отметить, что аббревиатура FCFS используется для этого алгоритма планирования вместо стандартной аббревиатуры FIFO для механизмов подобного типа для того, чтобы подчеркнуть, что организация готовых процессов в очередь FIFO возможна и при других алгоритмах планирования (например, для Round Robin – см. раздел «Round Robin (RR)»).

ное время выполнения оказывается равным (13 + 17 + 18)/3 = 16 единицам времени.

Если те же самые процессы расположены в порядке p_2, p_1, p_0, то картина их выполнения будет соответствовать рисунку 3.3. Время ожидания для процесса p_0 равняется 5 единицам времени, для процесса p_1 – 1 единице, для процесса p_2 – 0 единиц. Среднее время ожидания составит (5 + 1 + 0)/3 = 2 единицы времени. Это в 5 (!) раз меньше, чем в предыдущем случае. Полное время выполнения для процесса p_0 получается равным 18 единицам времени, для процесса p_1 – 5 единицам, для процесса p_2 – 1 единице. Среднее полное время выполнения составляет (18 + 5 + 1)/3 = 8 единиц времени, что в 2 раза меньше, чем при первой расстановке процессов.

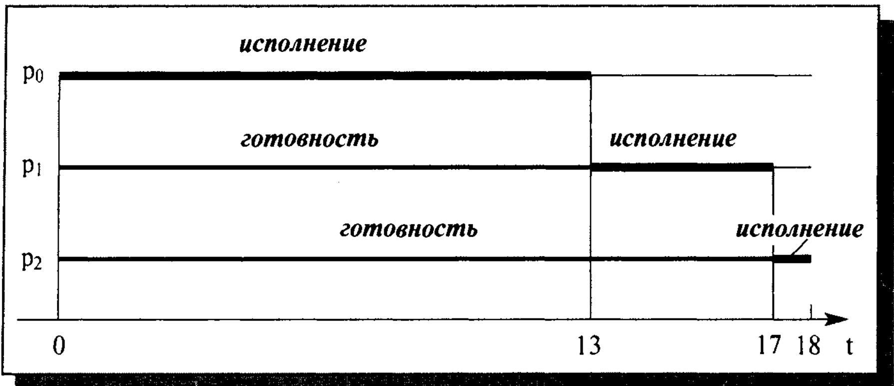


Рис. 3.2. Выполнение процессов при порядке p_0, p_1, p_2

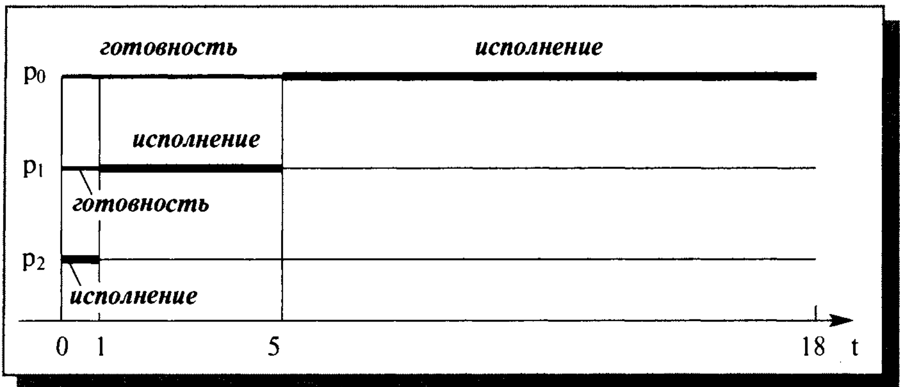


Рис. 3.3. Выполнение процессов при порядке p_2, p_1, p_0

Как мы видим, среднее время ожидания и среднее полное время выполнения для этого алгоритма существенно зависят от порядка расположения процессов в очереди. Если у нас есть процесс с длительным CPU burst, то короткие процессы, перешедшие в состояние ***готовность*** после длительного процесса, будут очень долго ждать начала выполнения. Поэтому алгоритм FCFS практически неприменим для систем разделения времени – слишком большим получается среднее время отклика в интерактивных процессах.

Round Robin (RR)

Модификацией алгоритма FCFS является алгоритм, получивший название Round Robin (Round Robin – это вид детской карусели в США) или сокращенно RR. По сути дела, это тот же самый алгоритм, только реализованный в режиме вытесняющего планирования. Можно представить себе все множество готовых процессов организованным циклически – процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный квант времени, обычно 10-100 миллисекунд (см. рис. 3.4). Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

Реализуется такой алгоритм так же, как и предыдущий, с помощью организации процессов, находящихся в состоянии ***готовность***, в очередь FIFO. Планировщик выбирает для очередного исполнения процесс, рас-

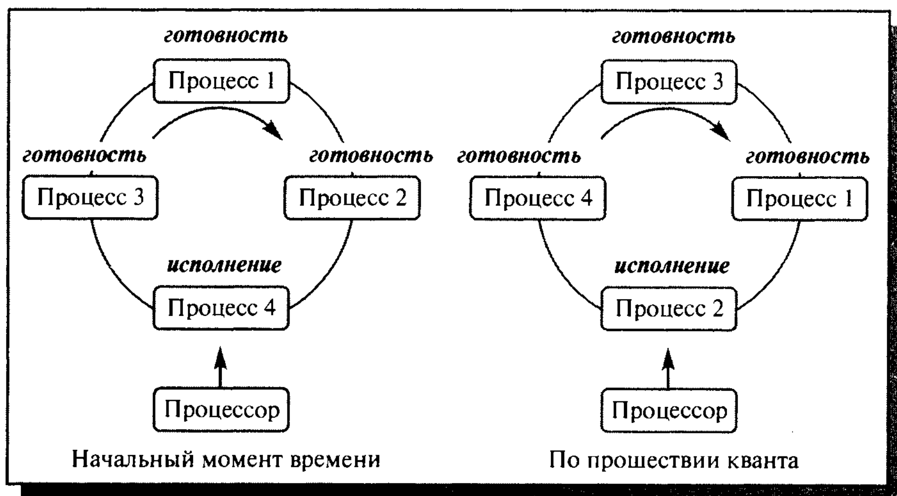


Рис. 3.4. Процессы на карусели

положенный в начале очереди, и устанавливает таймер для генерации прерывания по истечении определенного кванта времени. При выполнении процесса возможны два варианта:

- Время непрерывного использования процессора, необходимое процессу (остаток текущего CPU burst), меньше или равно продолжительности кванта времени. Тогда процесс по своей воле освобождает процессор до истечения кванта времени, на исполнение поступает новый процесс из начала очереди, и таймер начинает отсчет кванта заново.
- Продолжительность остатка текущего CPU burst процесса больше, чем квант времени. Тогда по истечении этого кванта процесс прерывается таймером и помещается в конец очереди процессов, готовых к исполнению, а процессор выделяется для использования процессу, находящемуся в ее начале.

Рассмотрим предыдущий пример с порядком процессов p_0, p_1, p_2 и величиной кванта времени, равной 4. Выполнение этих процессов иллюстрируется таблицей 3.2. Обозначение «И» используется в ней для процесса, находящегося в состоянии ***исполнение***, обозначение «Г» – для процессов в состоянии ***готовность***, пустые ячейки соответствуют завершившимся процессам. Состояния процессов показаны на протяжении соответствующей единицы времени, т. е. колонка с номером 1 соответствует промежутку времени от 0 до 1.

Таблица 3.2

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p_0	И	И	И	И	Г	Г	Г	Г	Г	И	И	И	И	И	И	И	И	И
p_1	Г	Г	Г	Г	И	И	И	И										
p_2	Г	Г	Г	Г	Г	Г	Г	Г	И									

Первым для исполнения выбирается процесс p_0. Продолжительность его CPU burst больше, чем величина кванта времени, и поэтому процесс исполняется до истечения кванта, т. е. в течение 4 единиц времени. После этого он помещается в конец очереди готовых к исполнению процессов, которая принимает вид p_1, p_2, p_0. Следующим начинает выполняться процесс p_1. Время его исполнения совпадает с величиной выделенного кванта, поэтому процесс работает до своего завершения. Теперь очередь процессов в состоянии ***готовность*** состоит из двух процессов — p_2 и p_0. Процессор выделяется процессу p_2. Он завершается до истечения отпущенного ему процессорного времени, и очередные кванты отмеряются процессу p_0 – единственному не закончившему к этому моменту свою работу. Время ожида-

ния для процесса p_0 (количество символов «Г» в соответствующей строке) составляет 5 единиц времени, для процесса p_1 – 4 единицы времени, для процесса p_2 — 8 единиц времени. Таким образом, среднее время ожидания для этого алгоритма получается равным (5 + 4 + 8)/3 = 5,6 (6) единицам времени. Полное время выполнения для процесса p_0 (количество непустых столбцов в соответствующей строке) составляет 18 единиц времени, для процесса p_1 – 8 единиц, для процесса p_2 – 9 единиц. Среднее полное время выполнения оказывается равным (18 + 8 + 9)/3 = 11,6 (6) единицам времени.

Легко увидеть, что среднее время ожидания и среднее полное время выполнения для обратного порядка процессов не отличаются от соответствующих времен для алгоритма FCFS и составляют 2 и 6 единиц времени соответственно.

На производительность алгоритма RR сильно влияет величина кванта времени. Рассмотрим тот же самый пример с порядком процессов p_0, p_1, p_2 для величины кванта времени, равной 1 (см. табл. 3.3). Время ожидания для процесса p_0 составит 5 единиц времени, для процесса p_1 – тоже 5 единиц, для процесса p_2 – 2 единицы. В этом случае среднее время ожидания получается равным (5 + 5 + 2)/3 = 4 единицам времени. Среднее полное время исполнения составит (18 + 9 + 3)/3 = 10 единиц времени.

Таблица 3.3

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p_0	И	Г	Г	И	Г	И	Г	И	Г	И	И	И	И	И	И	И	И	И
p_1	Г	И	Г	Г	И	Г	И	Г	И									
p_2	Г	Г	И															

При очень больших величинах кванта времени, когда каждый процесс успевает завершить свой CPU burst до возникновения прерывания по времени, алгоритм RR вырождается в алгоритм FCFS. При очень малых величинах создается иллюзия того, что каждый из n процессов работает на собственном виртуальном процессоре с производительностью ~ 1/n от производительности реального процессора. Правда, это справедливо лишь при теоретическом анализе при условии пренебрежения временами переключения контекста процессов. В реальных условиях при слишком малой величине кванта времени и, соответственно, слишком частом переключении контекста накладные расходы на переключение резко снижают производительность системы.

Shortest-Job-First (SJF)

При рассмотрении алгоритмов FCFS и RR мы видели, насколько существенным для них является порядок расположения процессов в очереди процессов, готовых к исполнению. Если короткие задачи расположены в очереди ближе к ее началу, то общая производительность этих алгоритмов значительно возрастает. Если бы мы знали время следующих CPU burst для процессов, находящихся в состоянии ***готовность***, то могли бы выбрать для исполнения не процесс из начала очереди, а процесс с минимальной длительностью CPU burst. Если же таких процессов два или больше, то для выбора одного из них можно использовать уже известный нам алгоритм FCFS. Квантование времени при этом не применяется. Описанный алгоритм получил название «кратчайшая работа первой» или *Shortest Job First (SJF)*.

SJF-алгоритм краткосрочного планирования может быть как вытесняющим, так и невытесняющим. При невытесняющем SJF-планировании процессор предоставляется избранному процессу на все необходимое ему время, независимо от событий, происходящих в вычислительной системе. При вытесняющем SJF-планировании учитывается появление новых процессов в очереди, готовых к исполнению (из числа вновь родившихся или разблокированных) во время работы выбранного процесса. Если CPU burst нового процесса меньше, чем остаток CPU burst у исполняющегося, то исполняющийся процесс вытесняется новым.

Рассмотрим пример работы невытесняющего алгоритма SJF. Пусть в состоянии ***готовность*** находятся четыре процесса — p_0, p_1, p_2 и p_3, для которых известны времена их очередных CPU burst. Эти времена приведены в таблице 3.4. Как и прежде, будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода и что временем переключения контекста можно пренебречь.

Таблица 3.4

Процесс	p_0	p_1	p_2	p_3
Продолжительность очередного CPU burst	5	3	7	1

При использовании невытесняющего алгоритма SJF первым для исполнения будет выбран процесс p_3, имеющий наименьшее значение продолжительности очередного CPU burst. После его завершения для исполнения выбирается процесс p_1, затем p_0 и, наконец, p_2. Эта картина отражена в таблице 3.5.

Таблица 3.5

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
p_0	Г	Г	Г	Г	И	И	И	И	И							
p_1	Г	И	И	И												
p_2	Г	Г	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И	И
p_3	И															

Как мы видим, среднее время ожидания для алгоритма SJF составляет (4 + 1 + 9 + 0)/4 = 3,5 единицы времени. Легко посчитать, что для алгоритма FCFS при порядке процессов p_0, p_1, p_2, p_3 эта величина будет равняться (0 + 5 + 8 + 15)/4 = 7 единицам времени, т. е. будет в два раза больше, чем для алгоритма SJF. Можно показать, что для заданного набора процессов (если в очереди не появляются новые процессы) алгоритм SJF является оптимальным с точки зрения минимизации среднего времени ожидания среди класса невытесняющих алгоритмов.

Для рассмотрения примера вытесняющего SJF планирования мы возьмем ряд процессов p_0, p_1, p_2 и p_3 с различными временами CPU burst и различными моментами их появления в очереди процессов, готовых к исполнению (см. табл. 3.6).

Таблица 3.6

Процесс	Время появления в очереди очередного CPU burst	Продолжительность
p_0	0	6
p_1	2	2
p_2	6	7
p_3	0	5

В начальный момент времени в состоянии ***готовность*** находятся только два процесса — p_0 и p_3. Меньшее время очередного CPU burst оказывается у процесса p_3, поэтому он и выбирается для исполнения (см. таблицу 3.7). По прошествии 2 единиц времени в систему поступает процесс p_1. Время его CPU burst меньше, чем остаток CPU burst у процесса p_3, который вытесняется из состояния ***исполнение*** и переводится в состояние ***готовность***. По прошествии еще 2 единиц времени процесс p_1 завершается, и для исполнения вновь выбирается процесс p_3. В момент времени

Таблица 3.7

Время	1	2	3	4	5	6	7	8	9	10
p_0	Г	Г	Г	Г	Г	Г	Г	И	И	И
p_1			И	И						
p_2							Г	Г	Г	Г
p_3	И	И	Г	Г	И	И	И			

Время	11	12	13	14	15	16	17	18	19	20
p_0	И	И	И							
p_1										
p_2	Г	Г	Г	И	И	И	И	И	И	И
p_3										

t = 6 в очереди процессов, готовых к исполнению, появляется процесс p_2, но поскольку ему для работы нужно 7 единиц времени, а процессу p_3 осталось трудиться всего 1 единицу времени, то процесс p_3 остается в состоянии ***исполнение***. После его завершения в момент времени t = 7 в очереди находятся процессы p_0 и p_2, из которых выбирается процесс p_0. Наконец, последним получит возможность выполняться процесс p_2.

Основную сложность при реализации алгоритма SJF представляет невозможность точного знания продолжительности очередного CPU burst для исполняющихся процессов. В пакетных системах количество процессорного времени, необходимое заданию для выполнения, указывает пользователь при формировании задания. Мы можем брать эту величину для осуществления долгосрочного SJF-планирования. Если пользователь укажет больше времени, чем ему нужно, он будет ждать результата дольше, чем мог бы, так как задание будет загружено в систему позже. Если же он укажет меньшее количество времени, задача может не досчитаться до конца. Таким образом, в пакетных системах решение задачи оценки времени использования процессора перекладывается на плечи пользователя. При краткосрочном планировании мы можем делать только прогноз длительности следующего CPU burst, исходя из предыстории работы процесса. Пусть $\tau(n)$ – величина n-го CPU burst $T(n + 1)$ – предсказываемое значение для n + 1-го CPU burst, α – некоторая величина в диапазоне от 0 до 1.

Определим рекуррентное соотношение

$$T(n+1)=\alpha\tau(n)+(1-\alpha)T(n)$$

T(0) положим произвольной константой. Первое слагаемое учитывает последнее поведение процесса, тогда как второе слагаемое учитывает его предысторию. При $\alpha = 0$ мы перестаем следить за последним поведением процесса, фактически полагая

$$T(n) = T(n+1) = ... = T(0)$$

т. е. оценивая все CPU burst одинаково, исходя из некоторого начального предположения.

Положив $\alpha = 1$, мы забываем о предыстории процесса. В этом случае мы полагаем, что время очередного CPU burst будет совпадать со временем последнего CPU burst

$$T(n+1)=\tau(n)$$

Обычно выбирают $\alpha = 1/2$ для равноценного учета последнего поведения и предыстории. Надо отметить, что такой выбор α удобен и для быстрой организации вычисления оценки $T(n + 1)$. Для подсчета новой оценки нужно взять старую оценку, сложить с измеренным временем CPU burst и полученную сумму разделить на 2, например, сдвинув ее на 1 бит вправо. Полученные оценки $T(n + 1)$ применяются как продолжительности очередных промежутков времени непрерывного использования процессора для краткосрочного SJF-планирования.

Гарантированное планирование

При интерактивной работе N пользователей в вычислительной системе можно применить алгоритм планирования, который гарантирует, что каждый из пользователей будет иметь в своем распоряжении ~1/N часть процессорного времени. Пронумеруем всех пользователей от 1 до N. Для каждого пользователя с номером i введем две величины: T_i – время нахождения пользователя в системе или, другими словами, длительность сеанса его общения с машиной и τ_i – суммарное процессорное время, уже выделенное всем его процессам в течение сеанса. Справедливым для пользователя было бы получение T_i/N процессорного времени. Если

$$\tau_i \ll \frac{T_i}{N}$$

то i-й пользователь несправедливо обделен процессорным временем. Если же

$$\tau_i >> \frac{T_i}{N}$$

то система явно благоволит к пользователю с номером i. Вычислим для процессов каждого пользователя значение коэффициента справедливости

$$\frac{\tau_i N}{T_i}$$

и будем предоставлять очередной квант времени готовому процессу с наименьшей величиной этого отношения. Предложенный алгоритм называют алгоритмом гарантированного планирования. К недостаткам этого алгоритма можно отнести невозможность предугадать поведение пользователей. Если некоторый пользователь отправится на пару часов пообедать и поспать, не прерывая сеанса работы, то по возвращении его процессы будут получать неоправданно много процессорного времени.

Приоритетное планирование

Алгоритмы SJF и гарантированного планирования представляют собой частные случаи приоритетного планирования. При приоритетном планировании каждому процессу присваивается определенное числовое значение – приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS. Для алгоритма SJF в качестве такого приоритета выступает оценка продолжительности следующего CPU burst. Чем меньше значение этой оценки, тем более высокий приоритет имеет процесс. Для алгоритма гарантированного планирования приоритетом служит вычисленный коэффициент справедливости. Чем он меньше, тем больше у процесса приоритет.

Алгоритмы назначения приоритетов процессов могут опираться как на внутренние параметры, связанные с происходящим внутри вычислительной системы, так и на внешние по отношению к ней. К внутренним параметрам относятся различные количественные и качественные характеристики процесса, такие как: ограничения по времени использования процессора, требования к размеру памяти, число открытых файлов и используемых устройств ввода-вывода, отношение средних продолжительностей I/O burst к CPU burst и т. д. Алгоритмы SJF и гарантированного планирова-

ния используют внутренние параметры. В качестве внешних параметров могут выступать важность процесса для достижения каких-либо целей, стоимость оплаченного процессорного времени и другие политические факторы. Высокий внешний приоритет может быть присвоен задаче лектора или того, кто заплатил $100 за работу в течение одного часа.

Планирование с использованием приоритетов может быть как вытесняющим, так и невытесняющим. При вытесняющем планировании процесс с более высоким приоритетом, появившийся в очереди готовых процессов, вытесняет исполняющийся процесс с более низким приоритетом. В случае невытесняющего планирования он просто становится в начало очереди готовых процессов. Давайте рассмотрим примеры использования различных режимов приоритетного планирования.

Пусть в очередь процессов, находящихся в состоянии ***готовность***, поступают те же процессы, что и в примере для вытесняющего алгоритма SJF, только им дополнительно еще присвоены приоритеты (см. табл. 3.8). В вычислительных системах не существует определенного соглашения, какое значение приоритета – 1 или 4 — считать более приоритетным. Во избежание путаницы, во всех наших примерах мы будем предполагать, что большее значение соответствует меньшему приоритету, т. е. наиболее приоритетным в нашем примере является процесс p_3, а наименее приоритетным – процесс p_0.

Таблица 3.8

Процесс	Время появления в очереди	Продолжительность очередного CPU burst	Приоритет
p_0	0	6	4
p_1	2	2	3
p_2	6	7	2
p_3	0	5	1

Как будут вести себя процессы при использовании невытесняющего приоритетного планирования? Первым для выполнения в момент времени $t = 0$ выбирается процесс p_3, как обладающий наивысшим приоритетом. После его завершения в момент времени $t = 5$ в очереди процессов, готовых к исполнению, окажутся два процесса — p_0 и p_1. Больший приоритет из них у процесса p_1, он и начнет выполняться (см. табл. 3.9). Затем в момент времени $t = 8$ для исполнения будет избран процесс p_2, и лишь потом – процесс p_0.

Таблица 3.9

Время	1	2	3	4	5	6	7	8	9	10
p_0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г
p_1			Г	Г	Г	И	И			
p_2							Г	И	И	И
p_3	И	И	И	И	И					

Время	11	12	13	14	15	16	17	18	19	20
p_0	Г	Г	Г	Г	И	И	И	И	И	И
p_1										
p_2	И	И	И	И						
p_3										

Иным будет предоставление процессора процессам в случае вытесняющего приоритетного планирования (см. табл. 3.10). Первым, как и в предыдущем случае, начнет исполняться процесс p_3, а по его окончании – процесс p_1. Однако в момент времени $t = 6$ он будет вытеснен процессом p_2 и продолжит свое выполнение только в момент времени $t = 13$. Последним, как и раньше, будет исполняться процесс p_0.

Таблица 3.10

Время	1	2	3	4	5	6	7	8	9	10
p_0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г
p_1			Г	Г	Г	И	Г	Г	Г	Г
p_2							И	И	И	И
p_3	И	И	И	И	И					

Время	11	12	13	14	15	16	17	18	19	20
p_0	Г	Г	Г	Г	И	И	И	И	И	И
p_1	Г	Г	Г	И						
p_2	И	И	И							
p_3										

В рассмотренном выше примере приоритеты процессов с течением времени не изменялись. Такие приоритеты принято называть статическими. Механизмы статической приоритетности легко реализовать, и они сопряжены с относительно небольшими издержками на выбор наиболее приоритетного процесса. Однако статические приоритеты не реагируют на изменения ситуации в вычислительной системе, которые могут сделать желательной корректировку порядка исполнения процессов. Более гибкими являются динамические приоритеты процессов, изменяющие свои значения по ходу исполнения процессов. Начальное значение динамического приоритета, присвоенное процессу, действует в течение лишь короткого периода времени, после чего ему назначается новое, более подходящее значение. Изменение динамического приоритета процесса является единственной операцией над процессами, которую мы до сих пор не рассмотрели. Как правило, изменение приоритета процессов проводится согласованно с совершением каких-либо других операций: при рождении нового процесса, при разблокировке или блокировании процесса, по истечении определенного кванта времени или по завершении процесса. Примерами алгоритмов с динамическими приоритетами являются алгоритм SJF и алгоритм гарантированного планирования. Схемы с динамической приоритетностью гораздо сложнее в реализации и связаны с большими издержками по сравнению со статическими схемами. Однако их использование предполагает, что эти издержки оправдываются улучшением работы системы.

Главная проблема приоритетного планирования заключается в том, что при ненадлежащем выборе механизма назначения и изменения приоритетов низкоприоритетные процессы могут не запускаться неопределенно долгое время. Обычно случается одно из двух — либо они все же дожидаются своей очереди на исполнение (в девять часов утра в воскресенье, когда все приличные программисты ложатся спать), либо вычислительную систему приходится выключать, и они теряются (при остановке IBM 7094 в Массачусетском технологическом институте в 1973 году были найдены процессы, запущенные в 1967 году и ни разу с тех пор не исполнявшиеся). Решение этой проблемы может быть достигнуто с помощью увеличения со временем значения приоритета процесса, находящегося в состоянии ***готовность***. Пусть изначально процессам присваиваются приоритеты от 128 до 255. Каждый раз по истечении определенного промежутка времени значения приоритетов готовых процессов уменьшаются на 1. Процессу, побывавшему в состоянии ***исполнение***, присваивается первоначальное значение приоритета. Даже такая грубая схема гарантирует, что любому процессу в разумные сроки будет предоставлено право на исполнение.

Многоуровневые очереди (Multilevel Queue)

Для систем, в которых процессы могут быть легко рассортированы по разным группам, был разработан другой класс алгоритмов планирования. Для каждой группы процессов создается своя очередь процессов, находящихся в состоянии ***готовность*** (см. рис. 3.5). Этим очередям приписываются фиксированные приоритеты. Например, приоритет очереди системных процессов устанавливается выше, чем приоритет очередей пользовательских процессов. А приоритет очереди процессов, запущенных студентами, ниже, чем для очереди процессов, запущенных преподавателями. Это значит, что ни один пользовательский процесс не будет выбран для исполнения, пока есть хоть один готовый системный процесс, и ни один студенческий процесс не получит в свое распоряжение процессор, если есть процессы преподавателей, готовые к исполнению. Внутри этих очередей для планирования могут применяться самые разные алгоритмы. Так, например, для больших счетных процессов, не требующих взаимодействия с пользователем (*фоновых* процессов), может использоваться алгоритм FCFS, а для интерактивных процессов – алгоритм RR. Подобный подход, получивший название многоуровневых очередей, повышает гибкость планирования: для процессов с различными характеристиками применяется наиболее подходящий им алгоритм.

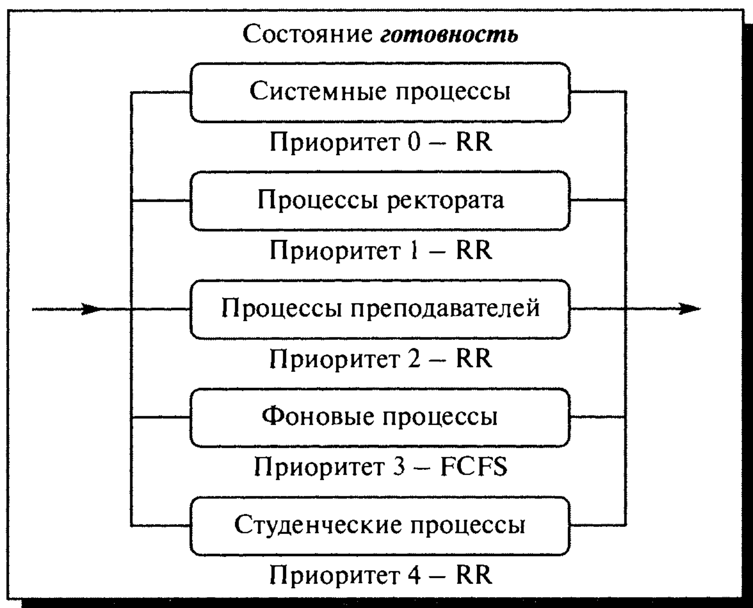


Рис. 3.5. Несколько очередей планирования

Многоуровневые очереди с обратной связью (Multilevel Feedback Queue)

Дальнейшим развитием алгоритма многоуровневых очередей является добавление к нему механизма обратной связи. Здесь процесс не постоянно приписан к определенной очереди, а может мигрировать из одной очереди в другую в зависимости от своего поведения.

Для простоты рассмотрим ситуацию, когда процессы в состоянии ***готовность*** организованы в 4 очереди, как на рисунке 3.6. Планирование процессов между очередями осуществляется на основе вытесняющего приоритетного механизма. Чем выше на рисунке располагается очередь, тем выше ее приоритет. Процессы в очереди 1 не могут исполняться, если в очереди 0 есть хотя бы один процесс. Процессы в очереди 2 не будут выбраны для выполнения, пока есть хоть один процесс в очередях 0 и 1. И наконец, процесс в очереди 3 может получить процессор в свое распоряжение только тогда, когда очереди 0, 1 и 2 пусты. Если при работе процесса появляется другой процесс в какой-либо более приоритетной очереди, исполняющийся процесс вытесняется новым. Планирование процессов внутри очередей 0–2 осуществляется с использованием алгоритма RR, планирование процессов в очереди 3 основывается на алгоритме FCFS.

Родившийся процесс поступает в очередь 0. При выборе на исполнение он получает в свое распоряжение квант времени размером 8 единиц. Если продолжительность его CPU burst меньше этого кванта времени, процесс остается в очереди 0. В противном случае он переходит в очередь 1. Для процессов из очереди 1 квант времени имеет величину 16. Если процесс не укладывается в это время, он переходит в очередь 2. Если укладывается – остается в очереди 1. В очереди 2 величина кванта времени составляет 32 единицы. Если для непрерывной работы процесса и этого мало, процесс поступает в очередь 3, для которой квантование времени не применяется и, при отсутствии готовых процессов в других очередях, может исполняться до окончания своего CPU burst. Чем больше значение продолжительности CPU burst, тем в менее приоритетную очередь попадает процесс, но тем на большее процессорное время он может рассчитывать. Таким образом, через некоторое время все процессы, требующие малого времени работы процессора, окажутся размещенными в высокоприоритетных очередях, а все процессы, требующие большого счета и с низкими запросами к времени отклика, – в низкоприоритетных.

Миграция процессов в обратном направлении может осуществляться по различным принципам. Например, после завершения ожидания ввода с клавиатуры процессы из очередей 1, 2 и 3 могут помещаться в очередь 0, после завершения дисковых операций ввода-вывода процессы из очередей 2 и 3 могут помещаться в очередь 1, а после завершения ожидания всех других

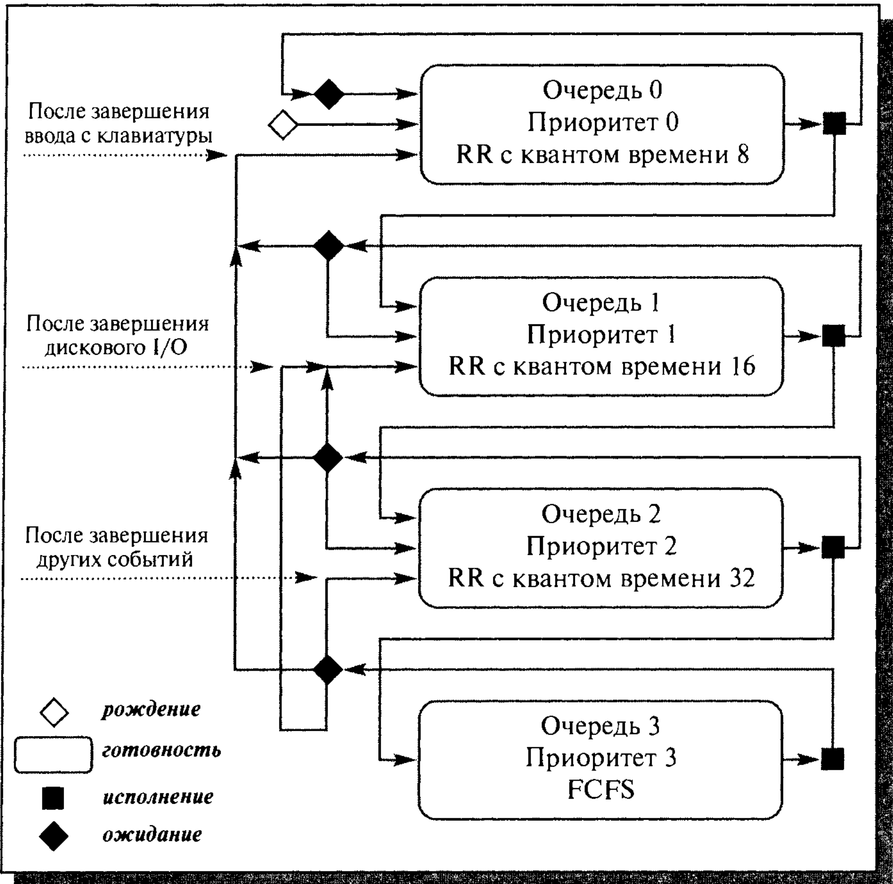


Рис. 3.6. Схема миграции процессов в многоуровневых очередях планирования с обратной связью. Вытеснение процессов более приоритетными процессами и завершение процессов на схеме не показано

событий – из очереди 3 в очередь 2. Перемещение процессов из очередей с низкими приоритетами в очереди с высокими приоритетами позволяет более полно учитывать изменение поведения процессов с течением времени.

Многоуровневые очереди с обратной связью представляют собой наиболее общий подход к планированию процессов из числа подходов, рассмотренных нами. Они наиболее трудны в реализации, но в то же время обладают наибольшей гибкостью. Понятно, что существует много других разновидностей такого способа планирования, помимо варианта, приведенного выше. Для полного описания их конкретного воплощения необходимо указать:

- Количество очередей для процессов, находящихся в состоянии ***готовность***.
- Алгоритм планирования, действующий между очередями.
- Алгоритмы планирования, действующие внутри очередей.
- Правила помещения родившегося процесса в одну из очередей.
- Правила перевода процессов из одной очереди в другую.

Изменяя какой-либо из перечисленных пунктов, мы можем существенно менять поведение вычислительной системы.

На этом мы прекращаем рассмотрение различных алгоритмов планирования процессов, ибо, как было сказано: «Нельзя объять необъятное».

Заключение

Одним из наиболее ограниченных ресурсов вычислительной системы является процессорное время. Для его распределения между многочисленными процессами в системе приходится применять процедуру планирования процессов. По степени длительности влияния планирования на поведение вычислительной системы различают краткосрочное, среднесрочное и долгосрочное планирование процессов. Конкретные алгоритмы планирования процессов зависят от поставленных целей, класса решаемых задач и опираются на статические и динамические параметры процессов и компьютерных систем. Различают вытесняющий и невытесняющий режимы планирования. При невытесняющем планировании исполняющийся процесс уступает процессор другому процессу только по собственному желанию, при вытесняющем планировании исполняющийся процесс может быть вытеснен из состояния исполнения помимо своей воли.

Простейшим алгоритмом планирования является невытесняющий алгоритм FCFS, который, однако, может существенно задерживать короткие процессы, не вовремя перешедшие в состояние ***готовность***. В системах разделения времени широкое распространение получила вытесняющая версия этого алгоритма – RR.

Среди всех невытесняющих алгоритмов оптимальным с точки зрения среднего времени ожидания процессов является алгоритм SJF. Существует и вытесняющий вариант этого алгоритма. В интерактивных системах часто используется алгоритм гарантированного планирования, обеспечивающий пользователям равные части процессорного времени.

Алгоритм SJF и алгоритм гарантированного планирования являются частными случаями планирования с использованием приоритетов. В более общих методах приоритетного планирования применяются многоуровневые очереди процессов, готовых к исполнению, и многоуровневые очереди с обратной связью. Будучи наиболее сложными в реализации, эти способы планирования обеспечивают гибкое поведение вычислительных систем и их адаптивность к решению задач разных классов.

Лекция 4. Кооперация процессов и основные аспекты ее логической организации

Одной из функций операционной системы является обеспечение санкционированного взаимодействия процессов. Лекция посвящена основам логической организации такого взаимодействия. Рассматривается расширение понятия процесс — нить исполнения (thread).

Ключевые слова: кооперация процессов, взаимодействующие процессы, независимые процессы, сигнальные средства связи, канальные средства связи, разделяемая память, прямая адресация, непрямая адресация, симплексная связь, полудуплексная связь, дуплексная связь, pipe, FIFO, именованный pipe, поток ввода-вывода, сообщения, нить исполнения, thread.

Взаимодействие процессов в вычислительной системе напоминает жизнь в коммунальной квартире. Постоянное ожидание в очереди к местам общего пользования (процессору) и ежедневная борьба за ресурсы (кто опять занял все конфорки на плите?). Для нормального функционирования процессов операционная система старается максимально обособить их друг от друга. Каждый процесс имеет собственное адресное пространство (каждая семья должна жить в отдельной комнате), нарушение которого, как правило, приводит к аварийной остановке процесса (вызов милиции). Каждому процессу по возможности предоставляются свои дополнительные ресурсы (каждая семья предпочитает иметь собственный холодильник). Тем не менее для решения некоторых задач (приготовление праздничного стола на всю квартиру) процессы могут объединять свои усилия. В настоящей лекции описываются причины взаимодействия процессов, способы их взаимодействия и возникающие при этом проблемы (попробуйте отремонтировать общую квартиру так, чтобы жильцы не перессорились друг с другом).

Взаимодействующие процессы

Для достижения поставленной цели различные процессы (возможно, даже принадлежащие разным пользователям) могут исполняться псевдопараллельно на одной вычислительной системе или параллельно на разных вычислительных системах, взаимодействуя между собой.

Для чего процессам нужно заниматься совместной деятельностью? Какие существуют причины для их кооперации?

- Повышение скорости работы. Пока один процесс ожидает наступления некоторого события (например, окончания операции ввода-

вывода), другие могут заниматься полезной работой, направленной на решение общей задачи. В многопроцессорных вычислительных системах программа разбивается на отдельные кусочки, каждый из которых будет исполняться на своем процессоре.

- Совместное использование данных. Различные процессы могут, к примеру, работать с одной и той же динамической базой данных или с разделяемым файлом, совместно изменяя их содержимое.
- Модульная конструкция какой-либо системы. Типичным примером может служить микроядерный способ построения операционной системы, когда различные ее части представляют собой отдельные процессы, взаимодействующие путем передачи сообщений через микроядро.
- Наконец, это может быть необходимо просто для удобства работы пользователя, желающего, например, редактировать и отлаживать программу одновременно. В этой ситуации процессы редактора и отладчика должны уметь взаимодействовать друг с другом.

Процессы не могут взаимодействовать, не общаясь, то есть не обмениваясь информацией. «Общение» процессов обычно приводит к изменению их поведения в зависимости от полученной информации. Если деятельность процессов остается неизменной при любой принятой ими информации, то это означает, что они на самом деле в «общении» не нуждаются. Процессы, которые влияют на поведение друг друга путем обмена информацией, принято называть *кооперативными* или *взаимодействующими* процессами, в отличие от *независимых* процессов, не оказывающих друг на друга никакого воздействия.

Различные процессы в вычислительной системе изначально представляют собой обособленные сущности. Работа одного процесса не должна приводить к нарушению работы другого процесса. Для этого, в частности, разделены их адресные пространства и системные ресурсы, и для обеспечения корректного взаимодействия процессов требуются специальные средства и действия операционной системы. Нельзя просто поместить значение, вычисленное в одном процессе, в область памяти, соответствующую переменной в другом процессе, не предприняв каких-либо дополнительных усилий. Давайте рассмотрим основные аспекты организации совместной работы процессов.

Категории средств обмена информацией

Процессы могут взаимодействовать друг с другом, только обмениваясь информацией. По объему передаваемой информации и степени возможного воздействия на поведение другого процесса все средства такого обмена можно разделить на три категории.

- *Сигнальные.* Передается минимальное количество информации – один бит, «да» или «нет». Используются, как правило, для извещения процесса о наступлении какого-либо события. Степень воздействия на поведение процесса, получившего информацию, минимальна. Все зависит от того, знает ли он, что означает полученный сигнал, надо ли на него реагировать и каким образом. Неправильная реакция на сигнал или его игнорирование могут привести к трагическим последствиям. Вспомним профессора Плейшнера из кинофильма «Семнадцать мгновений весны». Сигнал тревоги – цветочный горшок на подоконнике – был ему передан, но профессор проигнорировал его. И к чему это привело?
- *Канальные.* «Общение» процессов происходит через линии связи, предоставленные операционной системой, и напоминает общение людей по телефону, с помощью записок, писем или объявлений. Объем передаваемой информации в единицу времени ограничен пропускной способностью линий связи. С увеличением количества информации возрастает и возможность влияния на поведение другого процесса.
- *Разделяемая память.* Два или более процессов могут совместно использовать некоторую область адресного пространства. Созданием разделяемой памяти занимается операционная система (если, конечно, ее об этом попросят). «Общение» процессов напоминает совместное проживание студентов в одной комнате общежития. Возможность обмена информацией максимальна, как, впрочем, и влияние на поведение другого процесса, но требует повышенной осторожности (если вы переложили на другое место вещи вашего соседа по комнате, а часть из них еще и выбросили...). Использование разделяемой памяти для передачи/получения информации осуществляется с помощью средств обычных языков программирования, в то время как сигнальным и канальным средствам коммуникации для этого необходимы специальные системные вызовы. Разделяемая память представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.

Логическая организация механизма передачи информации

При рассмотрении любого из средств коммуникации нас будет интересовать не их физическая реализация (общая шина данных, прерывания, аппаратно разделяемая память и т. д.), а логическая, определяющая в конечном счете механизм их использования. Некоторые важные аспекты логической реализации являются общими для всех категорий средств связи, некоторые относятся к отдельным категориям. Давайте кратко охара-

ктеризуем основные вопросы, требующие разъяснения при изучении того или иного способа обмена информацией.

Как устанавливается связь?

Могу ли я использовать средство связи непосредственно для обмена информацией сразу после создания процесса или первоначально необходимо предпринять определенные действия для инициализации обмена? Например, для использования общей памяти различными процессами потребуется специальное обращение к операционной системе, которая выделит необходимую область адресного пространства. Но для передачи сигнала от одного процесса к другому никакая инициализация не нужна. В то же время передача информации по линиям связи может потребовать первоначального резервирования такой линии для процессов, желающих обменяться информацией.

К этому же вопросу тесно примыкает вопрос о способе адресации при использовании средства связи. Если я передаю некоторую информацию, я должен указать, куда я ее передаю. Если я желаю получить некоторую информацию, то мне нужно знать, откуда я могу ее получить.

Различают два способа адресации: *прямую* и *непрямую*. В случае прямой адресации взаимодействующие процессы непосредственно общаются друг с другом, при каждой операции обмена данными явно указывая имя или номер процесса, которому информация предназначена или от которого она должна быть получена. Если и процесс, от которого данные исходят, и процесс, принимающий данные, указывают имена своих партнеров по взаимодействию, то такая схема адресации называется *симметричной прямой адресацией*. **Ни один другой процесс не может вмешаться в процедуру симметричного прямого общения двух процессов, перехватить посланные или подменить ожидаемые данные**. Если только один из взаимодействующих процессов, например передающий, указывает имя своего партнера по кооперации, а второй процесс в качестве возможного партнера рассматривает любой процесс в системе, например ожидает получения информации от произвольного источника, то такая схема адресации называется *асимметричной прямой адресацией*.

При непрямой адресации данные помещаются передающим процессом в некоторый промежуточный объект для хранения данных, имеющий свой адрес, откуда они могут быть затем изъяты каким-либо другим процессом. Примером такого объекта может служить обычная доска объявлений или рекламная газета. При этом передающий процесс не знает, как именно идентифицируется процесс, который получит информацию, а принимающий процесс не имеет представления об идентификаторе процесса, от которого он должен ее получить.

При использовании прямой адресации связь между процессами в классической операционной системе устанавливается автоматически, без дополнительных инициализирующих действий. Единственное, что нужно для использования средства связи, — это знать, как идентифицируются процессы, участвующие в обмене данными.

При использовании непрямой адресации инициализация средства связи может и не требоваться. Информация, которой должен обладать процесс для взаимодействия с другими процессами, – это некий идентификатор промежуточного объекта для хранения данных, если он, конечно, не является единственным и неповторимым в вычислительной системе для всех процессов.

Информационная валентность процессов и средств связи

Следующий важный вопрос – это вопрос об информационной валентности связи. Слово «валентность» здесь использовано по аналогии с химией. Сколько процессов может быть одновременно ассоциировано с конкретным средством связи? Сколько таких средств связи может быть задействовано между двумя процессами?

Понятно, что при прямой адресации только одно фиксированное средство связи может быть задействовано для обмена данными между двумя процессами, и только эти два процесса могут быть ассоциированы с ним. При непрямой адресации может существовать более двух процессов, использующих один и тот же объект для данных, и более одного объекта может быть использовано двумя процессами.

К этой же группе вопросов следует отнести и вопрос о направленности связи. Является ли связь однонаправленной или двунаправленной? Под однонаправленной связью мы будем понимать связь, при которой каждый процесс, ассоциированный с ней, может использовать средство связи либо только для приема информации, либо только для ее передачи. При двунаправленной связи каждый процесс, участвующий в общении, может использовать связь и для приема, и для передачи данных. В коммуникационных системах принято называть однонаправленную связь *симплексной*, двунаправленную связь с поочередной передачей информации в разных направлениях – *полудуплексной*, а двунаправленную связь с возможностью одновременной передачи информации в разных направлениях – *дуплексной*. Прямая и непрямая адресация не имеет непосредственного отношения к направленности связи.

Особенности передачи информации с помощью линий связи

Как уже говорилось выше, передача информации между процессами посредством линий связи является достаточно безопасной по сравнению с

использованием разделяемой памяти и более информативной по сравнению с сигнальными средствами коммуникации. Кроме того, разделяемая память не может быть использована для связи процессов, функционирующих на различных вычислительных системах. Возможно, именно поэтому каналы связи из средств коммуникации процессов получили наибольшее распространение. Коснемся некоторых вопросов, связанных с логической реализацией канальных средств коммуникации.

Буферизация

Может ли линия связи сохранять информацию, переданную одним процессом, до ее получения другим процессом или помещения в промежуточный объект? Каков объем этой информации? Иными словами, речь идет о том, обладает ли канал связи *буфером* и каков объем этого буфера. Здесь можно выделить три принципиальных варианта:

1) Буфер нулевой емкости или отсутствует. Никакая информация не может сохраняться на линии связи. В этом случае процесс, посылающий информацию, должен ожидать, пока процесс, принимающий информацию, не соблаговолит ее получить, прежде чем заниматься своими дальнейшими делами (в реальности этот случай никогда не реализуется).
2) Буфер ограниченной емкости. Размер буфера равен n, то есть линия связи не может хранить до момента получения более чем n единиц информации. Если в момент передачи данных в буфере хватает места, то передающий процесс не должен ничего ожидать. Информация просто копируется в буфер. Если же в момент передачи данных буфер заполнен или места недостаточно, то необходимо задержать работу процесса отправителя до появления в буфере свободного пространства.
3) Буфер неограниченной емкости. Теоретически это возможно, но практически вряд ли реализуемо. Процесс, посылающий информацию, никогда не ждет окончания ее передачи и приема другим процессом.

При использовании канального средства связи с непрямой адресацией под емкостью буфера обычно понимается количество информации, которое может быть помещено в промежуточный объект для хранения данных.

Поток ввода/вывода и сообщения

Существует две модели передачи данных по каналам связи – поток *ввода-вывода* и *сообщения*. При передаче данных с помощью потоковой модели, операции передачи/приема информации вообще не интересуются содержимым данных. Процесс, прочитавший 100 байт из линии связи, не знает и не может знать, были ли они переданы одновременно, т. е. од-

ним куском или порциями по 20 байт, пришли они от одного процесса или от разных. Данные представляют собой простой поток байтов, без какой-либо их интерпретации со стороны системы. Примерами потоковых каналов связи могут служить pipe и FIFO, описанные ниже.

Одним из наиболее простых способов передачи информации между процессами по линиям связи является передача данных через pipe (канал, трубу или, как его еще называют в литературе, конвейер). Представим себе, что у нас есть некоторая труба в вычислительной системе, в один из концов которой процессы могут «сливать» информацию, а из другого конца принимать полученный поток. Такой способ реализует потоковую модель ввода/вывода. Информацией о расположении трубы в операционной системе обладает только процесс, создавший ее. Этой информацией он может поделиться исключительно со своими наследниками – процессами-детьми и их потомками. Поэтому использовать pipe для связи между собой могут только родственные процессы, имеющие общего предка, создавшего данный канал связи.

Если разрешить процессу, создавшему трубу, сообщать о ее местонахождении в системе другим процессам, сделав вход и выход трубы каким-либо образом видимыми для всех остальных, например, зарегистрировав ее в операционной системе под определенным именем, мы получим объект, который принято называть FIFO или именованный pipe. Именованный pipe может использоваться для организации связи между любыми процессами в системе.

В модели сообщений процессы налагают на передаваемые данные некоторую структуру. Весь поток информации они разделяют на отдельные сообщения, вводя между данными, по крайней мере, границы сообщений. Примером границ сообщений являются точки между предложениями в сплошном тексте или границы абзаца. Кроме того, к передаваемой информации могут быть присоединены указания на то, кем конкретное сообщение было послано и для кого оно предназначено. Примером указания отправителя могут служить подписи под эпиграфами в книге. Все сообщения могут иметь одинаковый фиксированный размер или могут быть переменной длины. В вычислительных системах используются разнообразные средства связи для передачи сообщений: очереди сообщений, sockets (гнезда) и т. д. Часть из них мы рассмотрим подробнее в дальнейшем, в частности очереди сообщений будут рассмотрены в лекции 6, а гнезда (иногда их еще называют по транслитерации английского названия – сокеты) в лекции 14.

И потоковые линии связи, и каналы сообщений всегда имеют буфер конечной длины. Когда мы будем говорить о емкости буфера для потоков данных, мы будем измерять ее в байтах. Когда мы будем говорить о емкости буфера для сообщений, мы будем измерять ее в сообщениях.

Надежность средств связи

Одним из существенных вопросов при рассмотрении всех категорий средств связи является вопрос об их надежности. Мы все знаем, как бывает тяжело расслышать собеседника по вечно трещащему телефону или разобрать, о чем сообщается в телеграмме: «Прибду пыездом в вонедельник 33 июня в 25.34. Пама».

Мы будем называть способ коммуникации надежным, если при обмене данными выполняются четыре условия:

1) Не происходит потери информации.
2) Не происходит повреждения информации.
3) Не появляется лишней информации.
4) Не нарушается порядок данных в процессе обмена.

Очевидно, что передача данных через разделяемую память является надежным способом связи. То, что мы сохранили в разделяемой памяти, будет считано другими процессами в первозданном виде, если, конечно, не произойдет сбоя в питании компьютера. Для других средств коммуникации, как видно из приведенных выше примеров, это не всегда верно.

Каким образом в вычислительных системах пытаются бороться с ненадежностью коммуникаций? Давайте рассмотрим возможные варианты на примере обмена данными через линию связи с помощью сообщений. Для обнаружения повреждения информации будем снабжать каждое передаваемое сообщение некоторой контрольной суммой, вычисленной по посланной информации. При приеме сообщения контрольную сумму будем вычислять заново и проверять ее соответствие пришедшему значению. Если данные не повреждены (контрольные суммы совпадают), то подтвердим правильность их получения. Если данные повреждены (контрольные суммы не совпадают), то сделаем вид, что сообщение к нам не поступило. Вместо контрольной суммы можно использовать специальное кодирование передаваемых данных с помощью кодов, исправляющих ошибки. Такое кодирование позволяет при числе искажений информации, не превышающем некоторого значения, восстановить первоначальные неискаженные данные. Если по прошествии некоторого интервала времени подтверждение правильности полученной информации не придет на передающий конец линии связи, будем считать информацию утерянной и пошлем ее повторно. Для того чтобы избежать двойного получения одной и той же информации, на приемном конце линии связи должен осуществляться соответствующий контроль. Для гарантии правильного порядка получения сообщений будем их нумеровать. При приеме сообщения с номером, не соответствующим ожидаемому, поступаем с ним как с утерянным и ждем сообщения с правильным номером.

Подобные действия могут быть возложены:

- на операционную систему;
- на процессы, обменивающиеся данными;
- совместно на систему и процессы, разделяя их ответственность. Операционная система может обнаруживать ошибки при передаче данных и извещать об этом взаимодействующие процессы для принятия ими решения о дальнейшем поведении.

Как завершается связь?

Наконец, важным вопросом при изучении средств обмена данными является вопрос прекращения обмена. Здесь нужно выделить два аспекта: требуются ли от процесса какие-либо специальные действия по прекращению использования средства коммуникации и влияет ли такое прекращение на поведение других процессов. Для способов связи, которые не подразумевали никаких инициализирующих действий, обычно ничего специального для окончания взаимодействия предпринимать не надо. Если же установление связи требовало некоторой инициализации, то, как правило, при ее завершении бывает необходимо выполнить ряд операций, например сообщить операционной системе об освобождении выделенного связного ресурса.

Если кооперативные процессы прекращают взаимодействие согласованно, то такое прекращение не влияет на их дальнейшее поведение. Иная картина наблюдается при несогласованном окончании связи одним из процессов. Если какой-либо из взаимодействующих процессов, не завершивших общение, находится в этот момент в состоянии ожидания получения данных либо попадает в такое состояние позже, то операционная система обязана предпринять некоторые действия для того, чтобы исключить вечное блокирование этого процесса. Обычно это либо прекращение работы ожидающего процесса, либо его извещение о том, что связи больше нет (например, с помощью передачи заранее определенного сигнала).

Нити исполнения

Рассмотренные выше аспекты логической реализации относятся к средствам связи, ориентированным на организацию взаимодействия различных процессов. Однако усилия, направленные на ускорение решения задач в рамках классических операционных систем, привели к появлению совершенно иных механизмов, к изменению самого понятия «процесс».

В свое время внедрение идеи мультипрограммирования позволило повысить пропускную способность компьютерных систем, т. е. уменьшить среднее время ожидания результатов работы процессов. Но любой отдельно взятый процесс в мультипрограммной системе никогда не мо-

жет быть выполнен быстрее, чем при работе в однопрограммном режиме на том же вычислительном комплексе. Тем не менее, если алгоритм решения задачи обладает определенным внутренним параллелизмом, мы могли бы ускорить его работу, организовав взаимодействие нескольких процессов. Рассмотрим следующий пример. Пусть у нас есть следующая программа на псевдоязыке программирования.

```
Ввести массив a
Ввести массив b
Ввести массив c
a = a + b
c = a + c
Вывести массив c
```

При выполнении такой программы в рамках одного процесса этот процесс четырежды будет блокироваться, ожидая окончания операций ввода-вывода. Но наш алгоритм обладает внутренним параллелизмом. Вычисление суммы массивов `a + b` можно было бы выполнять параллельно с ожиданием окончания операции ввода массива `c`.

```
Ввести массив a
Ожидание окончания операции ввода
Ввести массив b
Ожидание окончания операции ввода
Ввести массив c
Ожидание окончания операции ввода      a = a + b
c = a + c
Вывести массив c
Ожидание окончания операции вывода
```

Такое совмещение операций по времени можно было бы реализовать, используя два взаимодействующих процесса. Для простоты будем полагать, что средством коммуникации между ними служит разделяемая память. Тогда наши процессы могут выглядеть следующим образом:

```
Процесс 1                          Процесс 2

Ввести массив a                    Ожидание ввода
Ожидание окончания                   массивов a и b
  операции ввода
Ввести массив b
Ожидание окончания
```

```
  операции ввода
Ввести массив c
Ожидание окончания                a = a + b
  операции ввода
c = a + c
Вывести массив c
Ожидание окончания
  операции вывода
```

Казалось бы, мы предложили конкретный способ ускорения решения задачи. Однако в действительности дело обстоит не так просто. Второй процесс должен быть создан, оба процесса должны сообщить операционной системе, что им необходима память, которую они могли бы разделить с другим процессом, и, наконец, нельзя забывать о переключении контекста. Поэтому реальное поведение процессов будет выглядеть примерно так:

Процесс 1 Процесс 2

```
Создать процесс 2
             Переключение контекста
                                Выделение общей памяти
                                Ожидание ввода a и b
             Переключение контекста
Выделение общей памяти
Ввести массив a
Ожидание окончания
  операции ввода
Ввести массив b
Ожидание окончания
  операции ввода
Ввести массив c
Ожидание окончания
  операции ввода
             Переключение контекста
                                a = a + b
             Переключение контекста
c = a + c
Вывести массив c
Ожидание окончания
  операции вывода
```

Очевидно, что мы можем не только не выиграть во времени при решении задачи, но даже и проиграть, так как временные потери на созда-

ние процесса, выделение общей памяти и переключение контекста могут превысить выигрыш, полученный за счет совмещения операций.

Для того чтобы реализовать нашу идею, введем новую абстракцию внутри понятия «процесс» – *нить исполнения* или просто *нить* (в англоязычной литературе используется термин *thread*). Нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет собственный программный счетчик, свое содержимое регистров и свой стек. Теперь процесс представляется как совокупность взаимодействующих нитей и выделенных ему ресурсов. Процесс, содержащий всего одну нить исполнения, идентичен процессу в том смысле, который мы употребляли ранее. Для таких процессов мы в дальнейшем будем использовать термин «традиционный процесс». Иногда нити называют облегченными процессами или мини-процессами, так как во многих отношениях они подобны традиционным процессам. Нити, как и процессы, могут порождать нити-потомки, правда, только внутри своего процесса, и переходить из одного состояния в другое. Состояния нитей аналогичны состояниям традиционных процессов. Из состояния ***рождение*** процесс приходит содержащим всего одну нить исполнения. Другие нити процесса будут являться потомками этой нити-прародительницы. Мы можем считать, что процесс находится в состоянии ***готовность***, если хотя бы одна из его нитей находится в состоянии ***готовность*** и ни одна из нитей не находится в состоянии ***исполнение***. Мы можем считать, что процесс находится в состоянии ***исполнение***, если одна из его нитей находится в состоянии ***исполнение***. Процесс будет находиться в состоянии ***ожидание***, если все его нити находятся в состоянии ***ожидание***. Наконец, процесс находится в состоянии ***завершил исполнение***, если все его нити находятся в состоянии ***закончил исполнение***. Пока одна нить процесса заблокирована, другая нить того же процесса может выполняться. Нити разделяют процессор так же, как это делали традиционные процессы, в соответствии с рассмотренными алгоритмами планирования.

Поскольку нити одного процесса разделяют существенно больше ресурсов, чем различные процессы, то операции создания новой нити и переключения контекста между нитями одного процесса занимают значительно меньше времени, чем аналогичные операции для процессов в целом. Предложенная нами схема совмещения работы в терминах нитей одного процесса получает право на существование:

Нить 1 Нить 2

```
Создать нить 2
                  Переключение контекста нитей
                                        Ожидание ввода a и b
```

```
                      Переключение контекста нитей
Ввести массив a
Ожидание окончания
  операции ввода
Ввести массив b
Ожидание окончания
  операции ввода
Ввести массив c
Ожидание окончания
  операции ввода
                      Переключение контекста нитей
                                         a = a + b
                      Переключение контекста нитей
c = a + c
Вывести массив c
Ожидание окончания
  операции вывода
```

Различают операционные системы, поддерживающие нити на уровне ядра и на уровне библиотек. Все сказанное выше справедливо для операционных систем, поддерживающих нити на уровне ядра. В них планирование использования процессора происходит в терминах нитей, а управление памятью и другими системными ресурсами остается в терминах процессов. В операционных системах, поддерживающих нити на уровне библиотек пользователей, и планирование процессора, и управление системными ресурсами осуществляются в терминах процессов. Распределение использования процессора по нитям в рамках выделенного процессу временного интервала осуществляется средствами библиотеки. В подобных системах блокирование одной нити приводит к блокированию всего процесса, ибо ядро операционной системы не имеет представления о существовании нитей. По сути дела, в таких вычислительных системах просто имитируется наличие нитей исполнения.

Далее в этой части книги для простоты изложения мы будем использовать термин «процесс», хотя все сказанное будет относиться и к нитям исполнения.

Заключение

Для достижения поставленной цели различные процессы могут исполняться псевдопараллельно на одной вычислительной системе или параллельно на разных вычислительных системах, взаимодействуя между собой. Причинами для совместной деятельности процессов обычно явля-

ются: необходимость ускорения решения задачи, совместное использование обновляемых данных, удобство работы или модульный принцип построения программных комплексов. Процессы, которые влияют на поведение друг друга путем обмена информацией, называют кооперативными или взаимодействующими процессами, в отличие от независимых процессов, не оказывающих друг на друга никакого воздействия и ничего не знающих о взаимном существовании в вычислительной системе.

Для обеспечения корректного обмена информацией операционная система должна предоставить процессам специальные средства связи. По объему передаваемой информации и степени возможного воздействия на поведение процесса, получившего информацию, их можно разделить на три категории: сигнальные, канальные и разделяемую память. Через канальные средства коммуникации информация может передаваться в виде потока данных или в виде сообщений и накапливаться в буфере определенного размера. Для инициализации «общения» процессов и его прекращения могут потребоваться специальные действия со стороны операционной системы. Процессы, связываясь друг с другом, могут использовать непрямую, прямую симметричную и прямую асимметричную схемы адресации. Существуют одно- и двунаправленные средства передачи информации. Средства коммуникации обеспечивают надежную связь, если при общении процессов не происходит потери и повреждения информации, не появляется лишней информации, не нарушается порядок данных.

Усилия, направленные на ускорение решения задач в рамках классических операционных систем, привели к появлению новой абстракции внутри понятия «процесс» – нити исполнения или просто нити. Нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет собственный программный счетчик, свое содержимое регистров и свой стек. Теперь процесс представляется как совокупность взаимодействующих нитей и выделенных ему ресурсов. Нити могут порождать новые нити внутри своего процесса, они имеют состояния, аналогичные состояниям процесса, и могут переводиться операционной системой из одного состояния в другое. В системах, поддерживающих нити на уровне ядра, планирование использования процессора осуществляется в терминах нитей исполнения, а управление остальными системными ресурсами – в терминах процессов. Накладные расходы на создание новой нити и на переключение контекста между нитями одного процесса существенно меньше, чем на те же самые действия для процессов, что позволяет на однопроцессорной вычислительной системе ускорять решение задач с помощью организации работы нескольких взаимодействующих нитей.

Лекция 5. Алгоритмы синхронизации

Для корректного взаимодействия процессов недостаточно одних организационных усилий операционной системы. Необходимы определенные внутренние изменения в поведении процессов. В настоящей лекции рассматриваются вопросы, связанные с такими изменениями, приводятся программные алгоритмы корректной организации взаимодействия процессов.

Ключевые слова: атомарная операция, активность, interleaving (чередование), детерминированный и недетерминированный наборы процессов, условия Бернстайна, сотояние гонки (race condition), критическая секция, взаимоисключение (mutual exclusion), взаимосинхронизация, условие прогресса, условие ограниченного ожидания, алгоритм Петерсона, алгоритм булочной (Bakery algorithm), команды Test-and-Set, Swap.

В предыдущей лекции мы говорили о внешних проблемах кооперации, связанных с организацией взаимодействия процессов со стороны операционной системы. Предположим, что надежная связь процессов организована, и они умеют обмениваться информацией. Нужно ли нам предпринимать еще какие-либо действия для организации правильного решения задачи взаимодействующими процессами? Нужно ли изменять их внутреннее поведение? Разъяснению этих вопросов и посвящена данная лекция.

Interleaving, race condition и взаимоисключения

Давайте временно отвлечемся от операционных систем, процессов и нитей исполнения и поговорим о некоторых «активностях». Под активностями мы будем понимать последовательное выполнение ряда действий, направленных на достижение определенной цели. Активности могут иметь место в программном и техническом обеспечении, в обычной деятельности людей и животных. Мы будем разбивать активности на некоторые неделимые, или атомарные, операции. Например, активность «приготовление бутерброда» можно разбить на следующие атомарные операции:

1) Отрезать ломтик хлеба.
2) Отрезать ломтик колбасы.
3) Намазать ломтик хлеба маслом.
4) Положить ломтик колбасы на подготовленный ломтик хлеба.

Неделимые операции могут иметь внутренние невидимые действия (взять батон хлеба в левую руку, взять нож в правую руку, произвести отрезание). Мы же называем их неделимыми потому, что считаем выполняемыми за раз, без прерывания деятельности.

Пусть имеется две активности

```
P: a  b  c
Q: d  e  f
```

где a, b, c, d, e, f – атомарные операции. При последовательном выполнении активностей мы получаем такую последовательность атомарных действий:

```
PQ: a  b  c  d  e  f
```

Что произойдет при исполнении этих активностей псевдопараллельно, в режиме разделения времени? Активности могут расслоиться на неделимые операции с различным чередованием, то есть может произойти то, что в английском языке принято называть словом interleaving. Возможные варианты чередования:

```
a  b  c  d  e  f
a  b  d  c  e  f
a  b  d  e  c  f
a  b  d  e  f  c
a  d  b  c  e  f
......
d  e  f  a  b  c
```

Атомарные операции активностей могут чередоваться всевозможными различными способами с сохранением порядка расположения внутри активностей. Так как псевдопараллельное выполнение двух активностей приводит к чередованию их неделимых операций, результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения. Рассмотрим пример. Пусть у нас имеется две активности P и Q, состоящие из двух атомарных операций каждая:

```
P: x=2          Q: x=3
   y=x-1           y=x+1
```

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и y являются для активностей общими? Очевидно, что возможны четыре разных набора значений для пары (x, y): (3, 4), (2, 1), (2, 3) и (3, 2). Мы будем говорить, что набор активностей (например, программ) *детерминирован*, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает

одинаковые выходные данные. В противном случае он ***недетерминирован.*** Выше приведен пример недетерминированного набора программ. Понятно, что детерминированный набор активностей можно безбоязненно выполнять в режиме разделения времени. Для недетерминированного набора такое исполнение нежелательно.

Можно ли до получения результатов определить, является ли набор активностей детерминированным или нет? Для этого существуют достаточные условия Бернстайна. Изложим их применительно к программам с разделяемыми переменными.

Введем наборы входных и выходных переменных программы. Для каждой атомарной операции наборы входных и выходных переменных – это наборы переменных, которые атомарная операция считывает и записывает. Набор входных переменных программы `R(P)` (`R` от слова read) суть объединение наборов входных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы `W(P)` (`W` от слова write) суть объединение наборов выходных переменных для всех ее неделимых действий. Например, для программы

```
P: x=u+v
   y=x*w
```

получаем `R(P) = {u, v, x, w}`, `W(P) = {x, y}`. Заметим, что переменная `x` присутствует как в `R(P)`, так и в `W(P)`.

Теперь сформулируем условия Бернстайна:

Если для двух данных активностей `P` и `Q`:

- пересечение `W(P)` и `W(Q)` пусто,
- пересечение `W(P)` с `R(Q)` пусто,
- пересечение `R(P)` и `W(Q)` пусто,

тогда выполнение `P` и `Q` детерминировано.

Если эти условия не соблюдены, возможно, параллельное выполнение `P` и `Q` детерминировано, а может быть, и нет.

Случай двух активностей естественным образом обобщается на их большее количество.

Условия Бернстайна информативны, но слишком жестки. По сути дела, они требуют практически невзаимодействующих процессов. А нам хотелось бы, чтобы детерминированный набор образовывали активности, совместно использующие информацию и обменивающиеся ею. Для этого нам необходимо ограничить число возможных чередований атомарных операций, исключив некоторые чередования с помощью механизмов синхронизации выполнения программ, обеспечив тем самым упорядоченный доступ программ к некоторым данным.

Про недетерминированный набор программ (и активностей вообще) говорят, что он имеет *race condition* (состояние гонки, состояние состязания). В приведенном выше примере процессы состязаются за вычисление значений переменных x и y.

Задачу упорядоченного доступа к разделяемым данным (устранение race condition) в том случае, когда нам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, исключает для всех других процессов возможность одновременного общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется *взаимоисключением* (*mutual exclusion*). Если очередность доступа к разделяемым ресурсам важна для получения правильных результатов, то одними взаимоисключениями уже не обойтись, нужна взаимосинхронизация поведения программ.

Критическая секция

Важным понятием при изучении способов синхронизации процессов является понятие *критической секции (critical section)* программы. Критическая секция – это часть программы, исполнение которой может привести к возникновению race condition для определенного набора программ. Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом. Иными словами, необходимо обеспечить реализацию взаимоисключения для критических секций программ. Реализация взаимоисключения для критических секций программ с практической точки зрения означает, что по отношению к другим процессам, участвующим во взаимодействии, критическая секция начинает выполняться как атомарная операция. Давайте рассмотрим следующий пример, в котором псевдопараллельные взаимодействующие процессы представлены действиями различных студентов (таблица 5.1).

Здесь критический участок для каждого процесса – от операции «Обнаруживает, что хлеба нет» до операции «Возвращается в комнату» включительно. В результате отсутствия взаимоисключения мы из ситуации «Нет хлеба» попадаем в ситуацию «Слишком много хлеба». Если бы этот критический участок выполнялся как атомарная операция – «Достает два батона хлеба», то проблема образования излишков была бы снята.

Сделать процесс добывания хлеба атомарной операцией можно было бы следующим образом: перед началом этого процесса закрыть дверь изнутри на засов и уходить добывать хлеб через окно, а по окончании процесса вернуться в комнату через окно и отодвинуть засов. Тогда пока

Таблица 5.1

Время	Студент 1	Студент 2	Студент 3
17-05	Приходит в комнату		
17-07	Обнаруживает, что хлеба нет		
17-09	Уходит в магазин		
17-11		Приходит в комнату	
17-13		Обнаруживает, что хлеба нет	
17-15		Уходит в магазин	
17-17			Приходит в комнату
17-19			Обнаруживает, что хлеба нет
17-21			Уходит в магазин
17-23	Приходит в магазин		
17-25	Покупает 2 батона на всех		
17-27	Уходит из магазина		
17-29		Приходит в магазин	
17-31		Покупает 2 батона на всех	
17-33		Уходит из магазина	
17-35			Приходит в магазин
17-37			Покупает 2 батона на всех
17-39			Уходит из магазина
17-41	Возвращается в комнату		
17-43			
17-45			
17-47		Возвращается в комнату	
17-49			
17-51			
17-53			Возвращается в комнату

один студент добывает хлеб, все остальные находятся в состоянии ожидания под дверью (таблица 5.2).

Таблица 5.2

Время	Студент 1	Студент 2	Студент 3
17-05	Приходит в комнату		
17-07	Достает два батона хлеба		
17-43		Приходит в комнату	
17-47			Приходит в комнату

Итак, для решения задачи необходимо, чтобы в том случае, когда процесс находится в своем критическом участке, другие процессы не могли войти в свои критические участки. Мы видим, что критический участок должен сопровождаться прологом (entry section) – «закрыть дверь изнутри на засов» – и эпилогом (exit section) – «отодвинуть засов», которые не имеют отношения к активности одиночного процесса. Во время выполнения пролога процесс должен, в частности, получить разрешение на вход в критический участок, а во время выполнения эпилога – сообщить другим процессам о том, что он покинул критическую секцию.

В общем случае структура процесса, участвующего во взаимодействии, может быть представлена следующим образом

```
while (some condition) {
  entry section
    critical section
  exit section
    remainder section
}
```

Здесь под remainder section понимаются все атомарные операции, не входящие в критическую секцию.

Оставшаяся часть этой лекции посвящена различным способам программной организации пролога и эпилога критического участка в случае, когда очередность доступа к критическому участку не имеет значения.

Программные алгоритмы организации взаимодействия процессов

Требования, предъявляемые к алгоритмам

Организация взаимоисключения для критических участков, конечно, позволит избежать возникновения race condition, но не является достаточной для правильной и эффективной параллельной работы кооперативных процессов. Сформулируем пять условий, которые должны выполняться для хорошего программного алгоритма организации взаимодействия процессов, имеющих критические участки, если они могут проходить их в произвольном порядке:

1) Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения. При этом предполагается, что основные инструкции языка программирования (такие примитивные инструкции, как load, store, test) являются атомарными операциями.
2) Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.
3) Если процесс P_i исполняется в своем критическом участке, то не существует никаких других процессов, которые исполняются в соответствующих критических секциях. Это условие получило название *условия взаимоисключения* (mutual exclusion).
4) Процессы, которые находятся вне своих критических участков и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические участки. Если нет процессов в критических секциях и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в remainder section, должны принимать решение о том, какой процесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно долго. Это условие получило название *условия прогресса* (progress).
5) Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой критический участок. От того момента, когда процесс запросил разрешение на вход в критическую секцию, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои критические участки лишь ограниченное число раз. Это условие получило название *условия ограниченного ожидания* (bound waiting).

Надо заметить, что описание соответствующего алгоритма в нашем случае означает описание способа организации пролога и эпилога для критической секции.

Запрет прерываний

Наиболее простым решением поставленной задачи является следующая организация пролога и эпилога:

```
while (some condition) {
  запретить все прерывания
    critical section
  разрешить все прерывания
    remainder section
}
```

Поскольку выход процесса из состояния ***исполнение*** без его завершения осуществляется по прерыванию, внутри критической секции никто не может вмешаться в его работу. Однако такое решение может иметь далеко идущие последствия, поскольку позволяет процессу пользователя разрешать и запрещать прерывания во всей вычислительной системе. Допустим, что пользователь случайно или по злому умыслу запретил прерывания в системе и зациклил или завершил свой процесс. Без перезагрузки системы в такой ситуации не обойтись.

Тем не менее запрет и разрешение прерываний часто применяются как пролог и эпилог к критическим секциям внутри самой операционной системы, например при обновлении содержимого PCB.

Переменная-замок

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением равным `0`. Процесс может войти в критическую секцию только тогда, когда значение этой переменной-замка равно `0`, одновременно изменяя ее значение на `1` – закрывая замок. При выходе из критической секции процесс сбрасывает ее значение в `0` – замок открывается (как в случае с покупкой хлеба студентами в разделе «Критическая секция»):

```
shared int lock = 0; /* shared означает, что */
/* переменная является разделяемой */
```

```
while (some condition) {
  while(lock); lock = 1;
    critical section
  lock = 0;
    remainder section
}
```

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию взаимоисключения, так как действие `while(lock); lock = 1;` не является атомарным. Допустим, процесс P_0 протестировал значение переменной `lock` и принял решение двигаться дальше. В этот момент, еще до присвоения переменной `lock` значения 1, планировщик передал управление процессу P_1. Он тоже изучает содержимое переменной `lock` и тоже принимает решение войти в критический участок. Мы получаем два процесса, одновременно выполняющих свои критические секции.

Строгое чередование

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую для них обоих переменную с начальным значением `0`. Только теперь она будет играть не роль замка для критического участка, а явно указывать, кто может следующим войти в него. Для `i`-го процесса это выглядит так:

```
shared int turn = 0;

while (some condition) {
  while(turn != i);
    critical section
  turn = 1-i;
    remainder section
}
```

Очевидно, что взаимоисключение гарантируется, процессы входят в критическую секцию строго по очереди: P_0, P_1, P_0, P_1, P_0, ... Но наш алгоритм не удовлетворяет условию прогресса. Например, если значение `turn` равно `1`, и процесс P_0 готов войти в критический участок, он не может сделать этого, даже если процесс P_1 находится в *remainder section*.

Флаги готовности

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процесса имеют разделяемый массив флагов готовности входа процессов в критический участок:

```
shared int ready[2] = {0, 0};
```

Когда `i`-й процесс готов войти в критическую секцию, он присваивает элементу массива `ready[i]` значение равное `1`. После выхода из критической секции он, естественно, сбрасывает это значение в `0`. Процесс не входит в критическую секцию, если другой процесс уже готов к входу в критическую секцию или находится в ней:

```
while (some condition) {
   ready[i] = 1;
   while(ready[1-i]);
      critical section
   ready[i] = 0;
      remainder section
}
```

Полученный алгоритм обеспечивает взаимоисключение, позволяет процессу, готовому к входу в критический участок, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает условие прогресса. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания `ready[0]=1` планировщик передал процессор от процесса `0` процессу `1`, который также выполнил присваивание `ready[1]=1`. После этого оба процесса бесконечно долго ждут друг друга на входе в критическую секцию. Возникает ситуация, которую принято называть *тупиковой (deadlock)*. (Подробнее о тупиковых ситуациях рассказывается в лекции 7.)

Алгоритм Петерсона

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 году Петерсон (Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности:

```
shared int ready[2] = {0, 0};
shared int turn;
while (some condition) {
  ready[i] = 1;
  turn =1-i;
  while(ready[1-i] && turn == 1-i);
    critical section
  ready[i] = 0;
    remainder section
}
```

При исполнении пролога критической секции процесс P_i заявляет о своей готовности выполнить критический участок и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда следует после другого. Тем самым работу в критическом участке продолжит процесс, которому было сделано последнее предложение.

Давайте докажем, что все пять наших требований к алгоритму действительно удовлетворяются.

Удовлетворение требований 1 и 2 очевидно.

Докажем выполнение условия взаимоисключения методом от противного. Пусть оба процесса одновременно оказались внутри своих критических секций. Заметим, что процесс P_i может войти в критическую секцию, только если `ready[1-i] == 0` или `turn == i`. Заметим также, что если оба процесса выполняют свои критические секции одновременно, то значения флагов готовности для обоих процессов совпадают и равны 1. Могли ли оба процесса войти в критические секции из состояния, когда они оба одновременно находились в процессе выполнения цикла `while`? Нет, так как в этом случае переменная `turn` должна была бы одновременно иметь значения 0 и 1 (когда оба процесса выполняют цикл, значения переменных измениться не могут). Пусть процесс P_0 первым вошел в критический участок, тогда процесс P_1 должен был выполнить перед вхождением в цикл `while` по крайней мере один предваряющий оператор (`turn = 0;`). Однако после этого он не может выйти из цикла до окончания критического участка процесса P_0, так как при входе в цикл `ready[0] == 1` и `turn == 0`, и эти значения не могут измениться до тех пор, пока процесс P_0 не покинет свой критический участок. Мы пришли к противоречию. Следовательно, имеет место взаимоисключение.

Докажем выполнение условия прогресса. Возьмем, без ограничения общности, процесс P_0. Заметим, что он не может войти в свою критическую

секцию только при совместном выполнении условий ready[1] == 1 и turn == 1. Если процесс P_1 не готов к выполнению критического участка, то ready[1] == 0, и процесс P_0 может осуществить вход. Если процесс P_1 готов к выполнению критического участка, то ready[1] == 1 и переменная turn имеет значение 0 либо 1, позволяя процессу P_0 либо процессу P_1 начать выполнение критической секции. Если процесс P_1 завершил выполнение критического участка, то он сбросит свой флаг готовности ready[1] == 0, разрешая процессу P_0 приступить к выполнению критической работы. Таким образом, условие прогресса выполняется.

Отсюда же вытекает выполнение условия ограниченного ожидания. Так как в процессе ожидания разрешения на вход процесс P_0 не изменяет значения переменных, он сможет начать исполнение своего критического участка после не более чем одного прохода по критической секции процесса P_1.

Алгоритм булочной (Bakery algorithm)

Алгоритм Петерсона дает нам решение задачи корректной организации взаимодействия двух процессов. Давайте рассмотрим теперь соответствующий алгоритм для n взаимодействующих процессов, который получил название алгоритм булочной, хотя применительно к нашим условиям его следовало бы скорее назвать алгоритм регистратуры в поликлинике. Основная его идея выглядит так. Каждый вновь прибывающий клиент (он же процесс) получает талончик на обслуживание с номером. Клиент с наименьшим номером на талончике обслуживается следующим. К сожалению, из-за неатомарности операции вычисления следующего номера алгоритм булочной не гарантирует, что у всех процессов будут талончики с разными номерами. В случае равенства номеров на талончиках у двух или более клиентов первым обслуживается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом порядке). Разделяемые структуры данных для алгоритма – это два массива:

```
shared enum {false, true} choosing[n];
shared int number[n];
```

Изначально элементы этих массивов инициируются значениями false и 0 соответственно. Введем следующие обозначения:

```
(a,b) < (c,d), если a < c
или если a == c и b < d
max(a0, a1, ...., an) - это число k такое, что
k >= ai для всех i = 0, ...,n
```

Структура процесса P_i для алгоритма булочной приведена ниже:

```
while (some condition) {
  choosing[i] = true;
  number[i] = max(number[0], ..., number[n-1]) + 1;
  choosing[i] = false;
  for(j = 0; j < n; j++){
    while(choosing[j]);
    while(number[j] != 0 && (number[j],j) <
      (number[i],i));
  }
    critical section
  number[i] = 0;
    remainder section
}
```

Доказательство того, что этот алгоритм удовлетворяет условиям 1–5, выполните самостоятельно в качестве упражнения.

Аппаратная поддержка взаимоисключений

Наличие аппаратной поддержки взаимоисключений позволяет упростить алгоритмы и повысить их эффективность точно так же, как это происходит и в других областях программирования. Мы уже обращались к общепринятому hardware для решения задачи реализации взаимоисключений, когда говорили об использовании механизма запрета/разрешения прерываний.

Многие вычислительные системы помимо этого имеют специальные команды процессора, которые позволяют проверить и изменить значение машинного слова или поменять местами значения двух машинных слов в памяти, выполняя эти действия как атомарные операции. Давайте обсудим, как концепции таких команд могут использоваться для реализации взаимоисключений.

Команда Test-and-Set (проверить и присвоить 1)

О выполнении команды `Test-and-Set`, осуществляющей проверку значения логической переменной с одновременной установкой ее значения в `1`, можно думать как о выполнении функции:

```
int Test_and_Set (int *target){
  int tmp = *target;
```

```
    *target = 1;
    return tmp;
}
```

С использованием этой атомарной команды мы можем модифицировать наш алгоритм для переменной-замка, так чтобы он обеспечивал взаимоисключения

```
shared int lock = 0;

while (some condition) {
    while(Test_and_Set(&lock));
        critical section
    lock = 0;
        remainder section
}
```

К сожалению, даже в таком виде полученный алгоритм не удовлетворяет условию ограниченного ожидания для алгоритмов. Подумайте, как его следует изменить для соблюдения всех условий.

Команда Swap (обменять значения)

Выполнение команды `Swap`, обменивающей два значения, находящихся в памяти, можно проиллюстрировать следующей функцией:

```
void Swap (int *a, int *b){
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Применяя атомарную команду `Swap`, мы можем реализовать предыдущий алгоритм, введя дополнительную логическую переменную `key`, локальную для каждого процесса:

```
shared int lock = 0;
int key;

while (some condition) {
    key = 1;
    do Swap(&lock,&key);
```

```
    while (key);
      critical section
    lock = 0;
      remainder section
}
```

Заключение

Последовательное выполнение некоторых действий, направленных на достижение определенной цели, называется активностью. Активности состоят из атомарных операций, выполняемых неразрывно, как единичное целое. При исполнении нескольких активностей в псевдопараллельном режиме атомарные операции различных активностей могут перемешиваться между собой с соблюдением порядка следования внутри активностей. Это явление получило название *interleaving* (чередование). Если результаты выполнения нескольких активностей не зависят от варианта чередования, то такой набор активностей называется детерминированным. В противном случае он носит название недетерминированного. Существует достаточное условие Бернстайна для определения детерминированности набора активностей, но оно накладывает очень жесткие ограничения на набор, требуя практически не взаимодействующих активностей. Про недетерминированный набор активностей говорят, что он имеет *race condition* (условие гонки, состязания). Устранение *race condition* возможно при ограничении допустимых вариантов чередований атомарных операций с помощью синхронизации поведения активностей. Участки активностей, выполнение которых может привести к *race condition*, называют критическими участками. Необходимым условием для устранения *race condition* является организация взаимоисключения на критических участках: внутри соответствующих критических участков не может одновременно находиться более одной активности.

Для эффективных программных алгоритмов устранения *race condition* помимо условия взаимоисключения требуется выполнение следующих условий: алгоритмы не используют специальных команд процессора для организации взаимоисключений, алгоритмы ничего не знают о скоростях выполнения процессов, алгоритмы удовлетворяют условиям прогресса и ограниченного ожидания. Все эти условия выполняются в алгоритме Петерсона для двух процессов и алгоритме булочной – для нескольких процессов.

Применение специальных команд процессора, выполняющих ряд действий как атомарную операцию, – Test-and-Set, Swap – позволяет существенно упростить алгоритмы синхронизации процессов.

Лекция 6. Механизмы синхронизации

Для повышения производительности вычислительных систем и облегчения задачи программистов существуют специальные механизмы синхронизации. Описание некоторых из них – семафоров Дейкстры, мониторов Хора, очередей сообщений – приводится в этой лекции.

Ключевые слова: механизмы синхронизации, проблема producer-consumer, семафоры Дейкстры, мониторы Хора, условные переменные, очереди сообщений.

Рассмотренные в конце предыдущей лекции алгоритмы хотя и являются корректными, но достаточно громоздки и не обладают элегантностью. Более того, процедура ожидания входа в критический участок предполагает достаточно длительное вращение процесса в пустом цикле, то есть напрасную трату драгоценного времени процессора. Существуют и другие серьезные недостатки у алгоритмов, построенных средствами обычных языков программирования. Допустим, что в вычислительной системе находятся два взаимодействующих процесса: один из них – **H** – с высоким приоритетом, другой – **L** – с низким приоритетом. Пусть планировщик устроен так, что процесс с высоким приоритетом вытесняет низкоприоритетный процесс всякий раз, когда он готов к исполнению, и занимает процессор на все время своего CPU burst (если не появится процесс с еще большим приоритетом). Тогда в случае, если процесс **L** находится в своей критической секции, а процесс **H**, получив процессор, подошел ко входу в критическую область, мы получаем тупиковую ситуацию. Процесс **H** не может войти в критическую область, находясь в цикле, а процесс **L** не получает управления, чтобы покинуть критический участок.

Для того чтобы не допустить возникновения подобных проблем, были разработаны различные механизмы синхронизации более высокого уровня. Описанию ряда из них – семафоров, мониторов и сообщений – и посвящена данная лекция.

Семафоры

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал Дейкстра (Dijkstra) в 1965 году.

Концепция семафоров

Семафор представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исклю-

чением момента ее инициализации, может осуществляться только через две атомарные операции: P (от датского слова proberen – проверять) и V (от verhogen – увеличивать). Классическое определение этих операций выглядит следующим образом:

```
P(S):  пока S == 0 процесс блокируется;
   S = S - 1;
V(S):  S = S + 1;
```

Эта запись означает следующее: при выполнении операции P над семафором S сначала проверяется его значение. Если оно больше 0, то из S вычитается 1. Если оно меньше или равно 0, то процесс блокируется до тех пор, пока S не станет больше 0, после чего из S вычитается 1. При выполнении операции V над семафором S к его значению просто прибавляется 1. В момент создания семафор может быть инициализирован любым неотрицательным значением.

Подобные переменные-семафоры могут с успехом применяться для решения различных задач организации взаимодействия процессов. В ряде языков программирования они были непосредственно введены в синтаксис языка (например, в ALGOL-68), в других случаях реализуются с помощью специальных системных вызовов. Соответствующая целая переменная располагается внутри адресного пространства ядра операционной системы. Операционная система обеспечивает атомарность операций P и V, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов. Если при выполнении операции P заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

Решение проблемы producer-consumer с помощью семафоров

Одной из типовых задач, требующих организации взаимодействия процессов, является задача producer-consumer (производитель-потребитель). Пусть два процесса обмениваются информацией через буфер ограниченного размера. Производитель закладывает информацию в буфер, а потребитель извлекает ее оттуда. На этом уровне деятельность потребителя и производителя можно описать следующим образом:

```
Producer:   while(1) {
               produce_item;
               put_item;
            }
```

```
Consumer:   while(1) {
               get_item;
               consume_item;
            }
```

Если буфер заполнен, то производитель должен ждать, пока в нем появится место, чтобы положить туда новую порцию информации. Если буфер пуст, то потребитель должен дожидаться нового сообщения. Как можно реализовать эти условия с помощью семафоров? Возьмем три семафора: `empty`, `full` и `mutex`. Семафор `full` будем использовать для гарантии того, что потребитель будет ждать, пока в буфере появится информация. Семафор `empty` будем использовать для организации ожидания производителя при заполненном буфере, а семафор `mutex` – для организации взаимоисключения на критических участках, которыми являются действия `put_item` и `get_item` (операции «положить информацию» и «взять информацию» не могут пересекаться, так как в этом случае возникнет опасность искажения информации). Тогда решение задачи на С-подобном языке выглядит так:

```
   Semaphore mutex = 1;
   Semaphore empty = N; /* где N - емкость буфера*/
   Semaphore full = 0;
Producer:
   while(1) {
      produce_item;
      P(empty);
      P(mutex);
      put_item;
      V(mutex);
      V(full);
   }
Consumer:
   while(1) {
      P(full);
      P(mutex);
      get_item;
      V(mutex);
      V(empty);
      consume_item;
   }
```

Легко убедиться, что это действительно корректное решение поставленной задачи. Попутно заметим, что семафоры использовались здесь для достижения двух целей: организации взаимоисключения на критическом участке и взаимосинхронизации скорости работы процессов.

Мониторы

Хотя решение задачи producer-consumer с помощью семафоров выглядит достаточно изящно, программирование с их использованием требует повышенной осторожности и внимания, чем отчасти напоминает программирование на языке Ассемблера. Допустим, что в рассмотренном примере мы случайно поменяли местами операции `P`, сначала выполнив операцию для семафора `mutex`, а уже затем для семафоров `full` и `empty`. Допустим теперь, что потребитель, войдя в свой критический участок (`mutex` сброшен), обнаруживает, что буфер пуст. Он блокируется и начинает ждать появления сообщений. Но производитель не может войти в критический участок для передачи информации, так как тот заблокирован потребителем. Получаем тупиковую ситуацию.

В сложных программах произвести анализ правильности использования семафоров с карандашом в руках становится очень непросто. В то же время обычные способы отладки программ зачастую не дают результата, поскольку возникновение ошибок зависит от interleaving атомарных операций, и ошибки могут быть трудновоспроизводимы. Для того чтобы облегчить работу программистов, в 1974 году Хором (Hoare) был предложен механизм еще более высокого уровня, чем семафоры, получивший название мониторов. Мы с вами рассмотрим конструкцию, несколько отличающуюся от оригинальной.

Мониторы представляют собой тип данных, который может быть с успехом внедрен в объектно-ориентированные языки программирования. Монитор обладает собственными переменными, определяющими его состояние. Значения этих переменных извне могут быть изменены только с помощью вызова функций-методов, принадлежащих монитору. В свою очередь, эти функции-методы могут использовать в работе только данные, находящиеся внутри монитора, и свои параметры. На абстрактном уровне можно описать структуру монитора следующим образом:

```
monitor monitor_name {
   описание внутренних переменных ;

   void m1(...){...
   }
   void m2(...){...
```

```
    }
    ...
    void mn(...){...
    }

    {
      блок инициализации внутренних переменных ;
    }
}
```

Здесь функции m_1, ..., m_n представляют собой функции-методы монитора, а блок инициализации внутренних переменных содержит операции, которые выполняются один и только один раз: при создании монитора или при самом первом вызове любой функции-метода до ее исполнения.

Важной особенностью мониторов является то, что в любой момент времени только один процесс может быть активен, т. е. находиться в состоянии ***готовность*** или ***исполнение***, внутри данного монитора. Поскольку мониторы представляют собой особые конструкции языка программирования, компилятор может отличить вызов функции, принадлежащей монитору, от вызовов других функций и обработать его специальным образом, добавив к нему пролог и эпилог, реализующий взаимоисключение. Так как обязанность конструирования механизма взаимоисключений возложена на компилятор, а не на программиста, работа программиста при использовании мониторов существенно упрощается, а вероятность возникновения ошибок становится меньше.

Однако одних только взаимоисключений недостаточно для того, чтобы в полном объеме реализовать решение задач, возникающих при взаимодействии процессов. Нам нужны еще и средства организации очередности процессов, подобно семафорам `full` и `empty` в предыдущем примере. Для этого в мониторах было введено понятие *условных переменных* (*condition variables*)*, над которыми можно совершать две операции `wait` и `signal`, отчасти похожие на операции `P` и `V` над семафорами.

Если функция монитора не может выполняться дальше, пока не наступит некоторое событие, она выполняет операцию `wait` над какой-либо условной переменной. При этом процесс, выполнивший операцию `wait`, блокируется, становится неактивным, и другой процесс получает возможность войти в монитор.

Когда ожидаемое событие происходит, другой процесс внутри функции-метода совершает операцию `signal` над той же самой условной переменной. Это приводит к пробуждению ранее заблокированного процесса, и он становится активным. Если несколько процессов дожидались

* В некоторых русских изданиях их еще называют переменными состояния.

операции signal для этой переменной, то активным становится только один из них. Что можно предпринять для того, чтобы у нас не оказалось двух процессов, разбудившего и пробужденного, одновременно активных внутри монитора? Хор предложил, чтобы пробужденный процесс подавлял исполнение разбудившего процесса, пока он сам не покинет монитор. Несколько позже Хансен (Hansen) предложил другой механизм: разбудивший процесс покидает монитор немедленно после исполнения операции signal. Мы будем придерживаться подхода Хансена.

Необходимо отметить, что условные переменные, в отличие от семафоров Дейкстры, не умеют запоминать предысторию. Это означает, что операция signal всегда должна выполняться после операции wait. Если операция signal выполняется над условной переменной, с которой не связано ни одного заблокированного процесса, то информация о произошедшем событии будет утеряна. Следовательно, выполнение операции wait всегда будет приводить к блокированию процесса.

Давайте применим концепцию мониторов к решению задачи производитель-потребитель:

```
monitor ProducerConsumer {
   condition full, empty;
   int count;
   void put() {
      if(count == N) full.wait;
      put_item;
      count += 1;
      if(count == 1) empty.signal;
   }
   void get() {
      if (count == 0) empty.wait;
      get_item();
      count -= 1;
      if(count == N-1) full.signal;
   }
   {
      count = 0;
   }
}
Producer:
  while(1) {
     produce_item;
     ProducerConsumer.put();
  }
```

```
Consumer:
  while(1) {
    ProducerConsumer.get();
    consume_item;
  }
```

Легко убедиться, что приведенный пример действительно решает поставленную задачу.

Реализация мониторов требует разработки специальных языков программирования и компиляторов для них. Мониторы встречаются в таких языках, как параллельный Евклид, параллельный Паскаль, Java и т. д. Эмуляция мониторов с помощью системных вызовов для обычных широко используемых языков программирования не так проста, как эмуляция семафоров. Поэтому можно пользоваться еще одним механизмом со скрытыми взаимоисключениями, механизмом, о котором мы уже упоминали, – передачей сообщений.

Сообщения

Для прямой и непрямой адресации достаточно двух примитивов, чтобы описать передачу сообщений по линии связи – `send` и `receive`. В случае прямой адресации мы будем обозначать их так:

`send(P, message)` – послать сообщение message процессу `P`;
`receive(Q, message)` – получить сообщение message от процесса `Q`.

В случае непрямой адресации мы будем обозначать их так:

`send(A, message)` – послать сообщение message в почтовый ящик `A`;
`receive(A, message)` – получить сообщение message из почтового ящика `A`.

Примитивы `send` и `receive` уже имеют скрытый от наших глаз механизм взаимоисключения. Более того, в большинстве систем они уже имеют и скрытый механизм блокировки при чтении из пустого буфера и при записи в полностью заполненный буфер. Реализация решения задачи producer-consumer для таких примитивов становится неприлично тривиальной. Надо отметить, что, несмотря на простоту использования, передача сообщений в пределах одного компьютера происходит существенно медленнее, чем работа с семафорами и мониторами.

Эквивалентность семафоров, мониторов и сообщений

Мы рассмотрели три высокоуровневых механизма, использующихся для организации взаимодействия процессов. Можно показать, что в рамках одной вычислительной системы, когда процессы имеют возможность использовать разделяемую память, все они эквивалентны. Это означает, что любые два из предложенных механизмов могут быть реализованы на базе третьего, оставшегося механизма.

Реализация мониторов и передачи сообщений с помощью семафоров

Рассмотрим сначала, как реализовать мониторы с помощью семафоров. Для этого нам нужно уметь реализовывать взаимоисключения при входе в монитор и условные переменные. Возьмем семафор mutex с начальным значением 1 для реализации взаимоисключения при входе в монитор и по одному семафору c_i для каждой условной переменной. Кроме того, для каждой условной переменной заведем счетчик f_i для индикации наличия ожидающих процессов. Когда процесс входит в монитор, компилятор будет генерировать вызов функции monitor_enter, которая выполняет операцию P над семафором mutex для данного монитора. При нормальном выходе из монитора (то есть при выходе без вызова операции signal для условной переменной) компилятор будет генерировать вызов функции monitor_exit, которая выполняет операцию V над этим семафором.

Для выполнения операции wait над условной переменной компилятор будет генерировать вызов функции wait, которая выполняет операцию V для семафора mutex, разрешая другим процессам входить в монитор, и выполняет операцию P над соответствующим семафором c_i, блокируя вызвавший процесс. Для выполнения операции signal над условной переменной компилятор будет генерировать вызов функции signal_exit, которая выполняет операцию V над ассоциированным семафором c_i (если есть процессы, ожидающие соответствующего события), и выход из монитора, минуя функцию monitor_exit:

```
Semaphore mutex = 1;

void monitor_enter(){
   P(mutex);
}
```

```
void monitor_exit(){
   V(mutex);
}

Semaphore ci = 0;
int fi = 0;

void wait(i){
   fi=fi + 1;
   V(mutex);
   P(ci );
   fi=fi - 1;
}

void signal_exit(i){
   if (fi)V(ci );
   else V(mutex);
}
```

Заметим, что при выполнении функции `signal_exit`, если кто-либо ожидал этого события, процесс покидает монитор без увеличения значения семафора `mutex`, не разрешая тем самым всем процессам, кроме разбуженного, войти в монитор. Это увеличение совершит разбуженный процесс, когда покинет монитор обычным способом или когда выполнит новую операцию `wait` над какой-либо условной переменной.

Рассмотрим теперь, как реализовать передачу сообщений, используя семафоры. Для простоты опишем реализацию только одной очереди сообщений. Выделим в разделяемой памяти достаточно большую область под хранение сообщений, там же будем записывать, сколько пустых и заполненных ячеек находится в буфере, хранить ссылки на списки процессов, ожидающих чтения и памяти. Взаимоисключение при работе с разделяемой памятью будем обеспечивать семафором `mutex`. Также заведем по одному семафору c_i на взаимодействующий процесс, для того чтобы обеспечивать блокирование процесса при попытке чтения из пустого буфера или при попытке записи в переполненный буфер. Посмотрим, как такой механизм будет работать. Начнем с процесса, желающего получить сообщение.

Процесс-получатель с номером `i` прежде всего выполняет операцию `P(mutex)`, получая в монопольное владение разделяемую память. После чего он проверяет, есть ли в буфере сообщения. Если нет, то он заносит себя в список процессов, ожидающих сообщения, выполняет `V(mutex)` и `P(ci)`. Если сообщение в буфере есть, то он читает его, изменяет счет-

чики буфера и проверяет, есть ли процессы в списке процессов, жаждущих записи. Если таких процессов нет, то выполняется V(mutex), и процесс-получатель выходит из критической области. Если такой процесс есть (с номером j), то он удаляется из этого списка, выполняется V для его семафора c_j, и мы выходим из критического района. Проснувшийся процесс начинает выполняться в критическом районе, так как mutex у нас имеет значение 0 и никто более не может попасть в критический район. При выходе из критического района именно разбуженный процесс произведет вызов V(mutex).

Как строится работа процесса-отправителя с номером i? Процесс, посылающий сообщение, тоже ждет, пока он не сможет иметь монополию на использование разделяемой памяти, выполнив операцию P(mutex). Далее он проверяет, есть ли место в буфере, и если да, то помещает сообщение в буфер, изменяет счетчики и смотрит, есть ли процессы, ожидающие сообщения. Если нет, выполняет V(mutex) и выходит из критической области, если есть, «будит» один из них (с номером j), вызывая V(c_j), с одновременным удалением этого процесса из списка процессов, ожидающих сообщений, и выходит из критического региона без вызова V(mutex), предоставляя тем самым возможность разбуженному процессу прочитать сообщение. Если места в буфере нет, то процесс-отправитель заносит себя в очередь процессов, ожидающих возможности записи, и вызывает V(mutex) и P(c_i).

Реализация семафоров и передачи сообщений с помощью мониторов

Нам достаточно показать, что с помощью мониторов можно реализовать семафоры, так как получать из семафоров сообщения мы уже умеем.

Самый простой способ такой реализации выглядит следующим образом. Заведем внутри монитора переменную-счетчик, связанный с эмулируемым семафором список блокируемых процессов и по одной условной переменной на каждый процесс. При выполнении операции P над семафором вызывающий процесс проверяет значение счетчика. Если оно больше нуля, уменьшает его на 1 и выходит из монитора. Если оно равно 0, процесс добавляет себя в очередь процессов, ожидающих события, и выполняет операцию wait над своей условной переменной. При выполнении операции V над семафором процесс увеличивает значение счетчика, проверяет, есть ли процессы, ожидающие этого события, и если есть, удаляет один из них из списка и выполняет операцию signal для условной переменной, соответствующей процессу.

Реализация семафоров и мониторов с помощью очередей сообщений

Покажем, наконец, как реализовать семафоры с помощью очередей сообщений. Для этого воспользуемся более хитрой конструкцией, введя новый синхронизирующий процесс. Этот процесс имеет счетчик и очередь для процессов, ожидающих включения семафора. Для того чтобы выполнить операции P и V, процессы посылают синхронизирующему процессу сообщения, в которых указывают свои потребности, после чего ожидают получения подтверждения от синхронизирующего процесса.

После получения сообщения синхронизирующий процесс проверяет значение счетчика, чтобы выяснить, можно ли совершить требуемую операцию. Операция V всегда может быть выполнена, в то время как операция P может потребовать блокирования процесса. Если операция может быть совершена, то она выполняется, и синхронизирующий процесс посылает подтверждающее сообщение. Если процесс должен быть блокирован, то его идентификатор заносится в очередь блокированных процессов, и подтверждение не посылается. Позднее, когда какой-либо из других процессов выполнит операцию V, один из блокированных процессов удаляется из очереди ожидания и получает соответствующее подтверждение.

Поскольку мы показали ранее, как из семафоров построить мониторы, мы доказали эквивалентность мониторов, семафоров и сообщений.

Заключение

Для организации синхронизации процессов могут применяться специальные механизмы высокого уровня, блокирующие процесс, ожидающий входа в критическую секцию или наступления своей очереди для использования совместного ресурса. К таким механизмам относятся, например, семафоры, мониторы и сообщения. Все эти конструкции являются эквивалентными, т. е., используя любую из них, можно реализовать две оставшиеся.

Лекция 7. Тупики

В лекции рассматриваются вопросы взаимоблокировок, тупиковых ситуаций и «зависаний» системы.

Ключевые слова: взаимоблокировка, тупиковая ситуация (тупик), ресурс, условия возникновения тупиков, алгоритм банкира.

Введение

В предыдущих лекциях мы рассматривали способы синхронизации процессов, которые позволяют процессам успешно кооперироваться. Однако в некоторых случаях могут возникнуть непредвиденные затруднения. Предположим, что несколько процессов конкурируют за обладание конечным числом ресурсов. Если запрашиваемый процессом ресурс недоступен, ОС переводит данный процесс в состояние ожидания. В случае когда требуемый ресурс удерживается другим ожидающим процессом, первый процесс не сможет сменить свое состояние. Такая ситуация называется *тупиком* (deadlock). Говорят, что в мультипрограммной системе процесс находится в состоянии тупика, если он ожидает события, которое никогда не произойдет. Системная тупиковая ситуация, или «зависание системы», является следствием того, что один или более процессов находятся в состоянии тупика. Иногда подобные ситуации называют *взаимоблокировками*. В общем случае проблема тупиков эффективного решения не имеет.

Рассмотрим пример. Предположим, что два процесса осуществляют вывод с ленты на принтер. Один из них успел монополизировать ленту и претендует на принтер, а другой наоборот. После этого оба процесса оказываются заблокированными в ожидании второго ресурса (см. рис. 7.1).

Определение. Множество процессов находится в тупиковой ситуации, если каждый процесс из множества ожидает события, которое может вы-

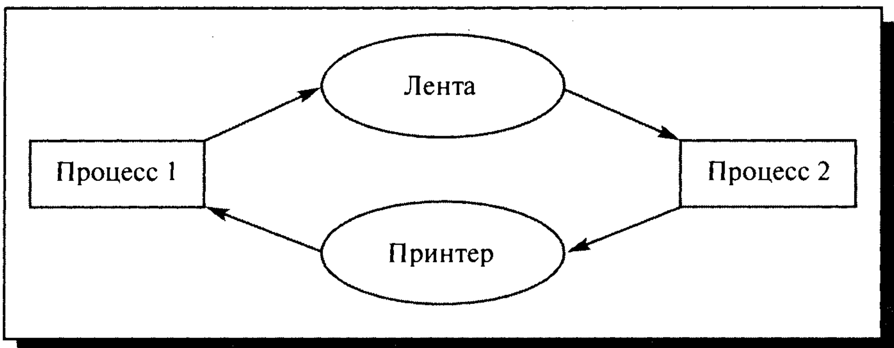


Рис. 7.1. Пример тупиковой ситуации

звать только другой процесс данного множества. Так как все процессы чего-то ожидают, то ни один из них не сможет инициировать событие, которое разбудило бы другого члена множества и, следовательно, все процессы будут спать вместе.

Выше приведен пример взаимоблокировки, возникающей при работе с так называемыми выделенными устройствами. Тупики, однако, могут иметь место и в других ситуациях. Например, в системах управления базами данных записи могут быть локализованы процессами, чтобы избежать состояния гонок (см. лекцию 5 «Алгоритмы синхронизации»). В этом случае может получиться так, что один из процессов заблокировал записи, необходимые другому процессу, и наоборот. Таким образом, тупики могут иметь место как на аппаратных, так и на программных ресурсах.

Тупики также могут быть вызваны ошибками программирования. Например, процесс может напрасно ждать открытия семафора, потому что в некорректно написанном приложении эту операцию забыли предусмотреть. Другой причиной бесконечного ожидания может быть дискриминационная политика по отношению к некоторым процессам. Однако чаще всего событие, которого ждет процесс в тупиковой ситуации, – освобождение ресурса, поэтому в дальнейшем будут рассмотрены методы борьбы с тупиками ресурсного типа.

Ресурсами могут быть как устройства, так и данные. Некоторые ресурсы допускают разделение между процессами, то есть являются *разделяемыми* ресурсами. Например, память, процессор, диски коллективно используются процессами. Другие не допускают разделения, то есть являются *выделенными*, например лентопротяжное устройство. К взаимоблокировке может привести использование как выделенных, так и разделяемых ресурсов. Например, чтение с разделяемого диска может одновременно осуществляться несколькими процессами, тогда как запись предполагает исключительный доступ к данным на диске. Можно считать, что часть диска, куда происходит запись, выделена конкретному процессу. Поэтому в дальнейшем мы будем исходить из предположения, что тупики связаны с выделенными ресурсами, то есть тупики возникают, когда процессу предоставляется эксклюзивный доступ к устройствам, файлам и другим ресурсам.

Традиционная последовательность событий при работе с ресурсом состоит из запроса, использования и освобождения ресурса. Тип запроса зависит от природы ресурса и от ОС. Запрос может быть явным, например специальный вызов request, или неявным – open для открытия файла. Обычно, если ресурс занят и запрос отклонен, запрашивающий процесс переходит в состояние ожидания.

Далее в данной лекции будут рассматриваться вопросы обнаружения, предотвращения, обхода тупиков и восстановления после тупиков. Как правило, борьба с тупиками – очень дорогостоящее мероприятие.

Тем не менее для ряда систем, например для систем реального времени, иного выхода нет.

Условия возникновения тупиков

Условия возникновения тупиков были сформулированы Коффманом, Элфиком и Шошани в 1970 году:

1) Условие взаимоисключения (Mutual exclusion). Одновременно использовать ресурс может только один процесс.
2) Условие ожидания ресурсов (Hold and wait). Процессы удерживают ресурсы, уже выделенные им, и могут запрашивать другие ресурсы.
3) Условие неперераспределяемости (No preemption). Ресурс, выделенный ранее, не может быть принудительно забран у процесса. Освобождены они могут быть только процессом, который их удерживает.
4) Условие кругового ожидания (Circular wait). Существует кольцевая цепь процессов, в которой каждый процесс ждет доступа к ресурсу, удерживаемому другим процессом цепи.

Для образования тупика необходимым и достаточным является выполнение **всех четырех** условий.

Обычно тупик моделируется циклом в графе, состоящем из узлов двух видов: прямоугольников – процессов и эллипсов – ресурсов, наподобие того, что изображен на рис. 7.1. Стрелки, направленные от ресурса к процессу, показывают, что ресурс выделен данному процессу. Стрелки, направленные от процесса к ресурсу, означают, что процесс запрашивает данный ресурс.

Основные направления борьбы с тупиками

Проблема тупиков инициировала много интересных исследований в области информатики. Очевидно, что условие циклического ожидания отличается от остальных. Первые три условия формируют правила, существующие в системе, тогда как четвертое условие описывает ситуацию, которая может сложиться при определенной неблагоприятной последовательности событий. Поэтому методы предотвращения взаимоблокировок ориентированы главным образом на нарушение первых трех условий путем введения ряда ограничений на поведение процессов и способы распределения ресурсов. Методы обнаружения и устранения менее консервативны и сводятся к поиску и разрыву цикла ожидания ресурсов.

Итак, основные направления борьбы с тупиками:

- Игнорирование проблемы в целом.
- Предотвращение тупиков.
- Обнаружение тупиков.
- Восстановление после тупиков.

Игнорирование проблемы тупиков

Простейший подход – не замечать проблему тупиков. Для того чтобы принять такое решение, необходимо оценить вероятность возникновения взаимоблокировки и сравнить ее с вероятностью ущерба от других отказов аппаратного и программного обеспечения. Проектировщики обычно не желают жертвовать производительностью системы или удобством пользователей для внедрения сложных и дорогостоящих средств борьбы с тупиками.

Любая ОС, имеющая в ядре ряд массивов фиксированной размерности, потенциально страдает от тупиков, даже если они не обнаружены. Таблица открытых файлов, таблица процессов, фактически каждая таблица являются ограниченными ресурсами. Заполнение всех записей таблицы процессов может привести к тому, что очередной запрос на создание процесса может быть отклонен. При неблагоприятном стечении обстоятельств несколько процессов могут выдать такой запрос одновременно и оказаться в тупике. Следует ли отказываться от вызова CreateProcess, чтобы решить эту проблему?

Подход большинства популярных ОС (Unix, Windows и др.) состоит в том, чтобы игнорировать данную проблему в предположении, что маловероятный случайный тупик предпочтительнее, чем нелепые правила, заставляющие пользователей ограничивать число процессов, открытых файлов и т. п. Сталкиваясь с нежелательным выбором между строгостью и удобством, трудно найти решение, которое устраивало бы всех.

Способы предотвращения тупиков

Цель предотвращения тупиков – обеспечить условия, исключающие возможность возникновения тупиковых ситуаций. Большинство методов связано с предотвращением одного из условий возникновения взаимоблокировки.

Система, предоставляя ресурс в распоряжение процесса, должна принять решение, безопасно это или нет. Возникает вопрос: есть ли такой алгоритм, который помогает всегда избегать тупиков и делать правильный выбор. Ответ – да, мы можем избегать тупиков, но только если определенная информация известна заранее.

Способы предотвращения тупиков путем тщательного распределения ресурсов. Алгоритм банкира

Можно избежать взаимоблокировки, если распределять ресурсы, придерживаясь определенных правил. Среди такого рода алгоритмов

наиболее известен алгоритм банкира, предложенный Дейкстрой, который базируется на так называемых *безопасных* или *надежных* состояниях (safe state). Безопасное состояние – это такое состояние, для которого имеется по крайней мере одна последовательность событий, которая не приведет к взаимоблокировке. Модель алгоритма основана на действиях банкира, который, имея в наличии капитал, выдает кредиты.

Суть алгоритма состоит в следующем:

- Предположим, что у системы в наличии n устройств, например, лент.
- ОС принимает запрос от пользовательского процесса, если его максимальная потребность не превышает n.
- Пользователь гарантирует, что если ОС в состоянии удовлетворить его запрос, то все устройства будут возвращены системе в течение конечного времени.
- Текущее состояние системы называется *надежным*, если ОС может обеспечить всем процессам их выполнение в течение конечного времени.
- В соответствии с алгоритмом банкира выделение устройств возможно, только если состояние системы остается надежным.

Рассмотрим пример надежного состояния для системы с 3 пользователями и 11 устройствами, где 9 устройств задействовано, а 2 имеется в резерве. Пусть текущая ситуация такова:

Таблица 7.1

Пользователи	Максимальная потребность в ресурсах	Выделенное пользователям количество ресурсов
Первый	9	6
Второй	10	2
Третий	3	1

Данное состояние надежно. Последующие действия системы могут быть таковы. Вначале удовлетворить запросы третьего пользователя, затем дождаться, когда он закончит работу и освободит свои три устройства. Затем можно обслужить первого и второго пользователей. То есть система удовлетворяет только те запросы, которые оставляют ее в надежном состоянии, и отклоняет остальные.

Термин *ненадежное* состояние не предполагает, что обязательно возникнут тупики. Он лишь говорит о том, что в случае неблагоприятной последовательности событий система может зайти в тупик.

Данный алгоритм обладает тем достоинством, что при его использовании нет необходимости в перераспределении ресурсов и откате процес-

сов назад. Однако использование этого метода требует выполнения ряда условий:

- Число пользователей и число ресурсов фиксировано.
- Число работающих пользователей должно оставаться постоянным.
- Алгоритм требует, чтобы клиенты гарантированно возвращали ресурсы.
- Должны быть заранее указаны максимальные требования процессов к ресурсам. Чаще всего данная информация отсутствует.

Наличие таких жестких и зачастую неприемлемых требований может склонить разработчиков к выбору других решений проблемы взаимоблокировки.

Предотвращение тупиков за счет нарушения условий возникновения тупиков

В отсутствие информации о будущих запросах единственный способ избежать взаимоблокировки – добиться невыполнения хотя бы одного из условий раздела «Условия возникновения тупиков».

Нарушение условия взаимоисключения

В общем случае избежать взаимоисключений невозможно. Доступ к некоторым ресурсам должен быть исключительным. Тем не менее некоторые устройства удается обобществить. В качестве примера рассмотрим принтер. Известно, что пытаться осуществлять вывод на принтер могут несколько процессов. Во избежание хаоса организуют промежуточное формирование всех выходных данных процесса на диске, то есть разделяемом устройстве. Лишь один системный процесс, называемый сервисом или демоном принтера, отвечающий за вывод документов на печать по мере освобождения принтера, реально с ним взаимодействует. Эта схема называется спулингом (spooling). Таким образом, принтер становится разделяемым устройством, и тупик для него устранен.

К сожалению, не для всех устройств и не для всех данных можно организовать спулинг. Неприятным побочным следствием такой модели может быть потенциальная тупиковая ситуация из-за конкуренции за дисковое пространство для буфера спулинга. Тем не менее в той или иной форме эта идея применяется часто.

Нарушение условия ожидания дополнительных ресурсов

Условия ожидания ресурсов можно избежать, потребовав выполнения стратегии двухфазного захвата.

- В первой фазе процесс должен запрашивать все необходимые ему ресурсы сразу. До тех пор пока они не предоставлены, процесс не может продолжать выполнение.
- Если в первой фазе некоторые ресурсы, которые были нужны данному процессору, уже заняты другими процессами, он освобождает все ресурсы, которые были ему выделены, и пытается повторить первую фазу.

В известном смысле этот подход напоминает требование захвата всех ресурсов заранее. Естественно, что только специально организованные программы могут быть приостановлены в течение первой фазы и рестартованы впоследствии.

Таким образом, один из способов – заставить все процессы затребовать нужные им ресурсы перед выполнением («все или ничего»). Если система в состоянии выделить процессу все необходимое, он может работать до завершения. Если хотя бы один из ресурсов занят, процесс будет ждать.

Данное решение применяется в пакетных *мэйнфреймах* (mainframe), которые требуют от пользователей перечислить все необходимые его программе ресурсы. Другим примером может служить механизм двухфазной локализации записей в СУБД. Однако в целом подобный подход не слишком привлекателен и приводит к неэффективному использованию компьютера. Как уже отмечалось, перечень будущих запросов к ресурсам редко удается спрогнозировать. Если такая информация есть, то можно воспользоваться алгоритмом банкира. Заметим также, что описываемый подход противоречит парадигме модульности в программировании, поскольку приложение должно знать о предполагаемых запросах к ресурсам во всех модулях.

Нарушение принципа отсутствия перераспределения

Если бы можно было отбирать ресурсы у удерживающих их процессов до завершения этих процессов, то удалось бы добиться невыполнения третьего условия возникновения тупиков. Перечислим минусы данного подхода.

Во-первых, отбирать у процессов можно только те ресурсы, состояние которых легко сохранить, а позже восстановить, например состояние процессора. Во-вторых, если процесс в течение некоторого времени использует определенные ресурсы, а затем освобождает эти ресурсы, он может потерять результаты работы, проделанной до настоящего момента. Наконец, следствием данной схемы может быть дискриминация отдельных процессов, у которых постоянно отбирают ресурсы.

Весь вопрос в цене подобного решения, которая может быть слишком высокой, если необходимость отбирать ресурсы возникает часто.

Нарушение условия кругового ожидания

Трудно предложить разумную стратегию, чтобы избежать последнего условия из раздела «Условия возникновения тупиков» – циклического ожидания.

Один из способов – упорядочить ресурсы. Например, можно присвоить всем ресурсам уникальные номера и потребовать, чтобы процессы запрашивали ресурсы в порядке их возрастания. Тогда круговое ожидание возникнуть не может. После последнего запроса и освобождения всех ресурсов можно разрешить процессу опять осуществить первый запрос. Очевидно, что практически невозможно найти порядок, который удовлетворит всех.

Один из немногих примеров упорядочивания ресурсов – создание иерархии спин-блокировок в Windows 2000. Спин-блокировка – простейший способ синхронизации (вопросы синхронизации процессов рассмотрены в соответствующей лекции). Спин-блокировка может быть захвачена и освобождена процессом. Классическая тупиковая ситуация возникает, когда процесс P_1 захватывает спин-блокировку S_1 и претендует на спин-блокировку S_2, а процесс P_2, захватывает спин-блокировку S_2 и хочет дополнительно захватить спин-блокировку S_1. Чтобы этого избежать, все спин-блокировки помещаются в упорядоченный список. Захват может осуществляться только в порядке, указанном в списке.

Другой способ атаки условия кругового ожидания – действовать в соответствии с правилом, согласно которому каждый процесс может иметь только один ресурс в каждый момент времени. Если нужен второй ресурс – освободи первый. Очевидно, что для многих процессов это неприемлемо.

Таким образом, технология предотвращения циклического ожидания, как правило, неэффективна и может без необходимости закрывать доступ к ресурсам.

Обнаружение тупиков

Обнаружение взаимоблокировки сводится к фиксации тупиковой ситуации и выявлению вовлеченных в нее процессов. Для этого производится проверка наличия циклического ожидания в случаях, когда выполнены первые три условия возникновения тупика. Методы обнаружения активно используют графы распределения ресурсов.

Рассмотрим модельную ситуацию:

- Процесс P_1 ожидает ресурс R_1.

- Процесс P_2 удерживает ресурс P_2 и ожидает ресурс R_1.
- Процесс P_3 удерживает ресурс R_1 и ожидает ресурс R_3.
- Процесс P_4 ожидает ресурс R_2.
- Процесс P_5 удерживает ресурс R_3 и ожидает ресурс R_2.

Вопрос состоит в том, является ли данная ситуация тупиковой, и если да, то какие процессы в ней участвуют. Для ответа на этот вопрос можно сконструировать граф ресурсов, как показано на рис. 7.2. Из рисунка видно, что имеется цикл, моделирующий условие кругового ожидания, и что процессы P_2, P_3, P_5 (а может быть, и другие) находятся в тупиковой ситуации.

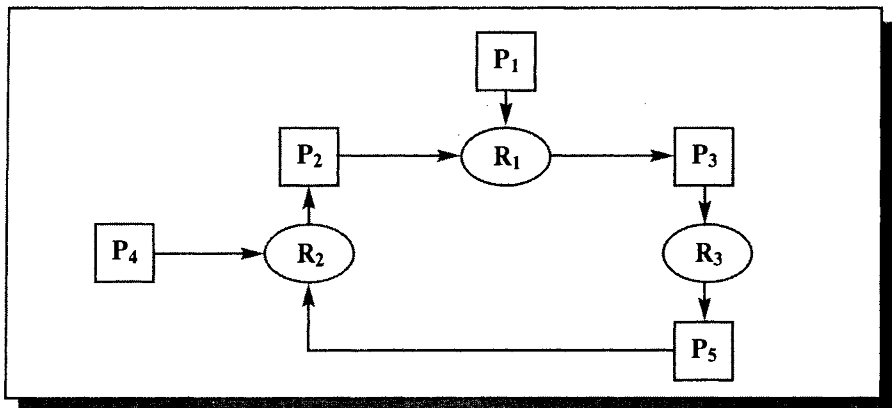


Рис. 7.2. Граф ресурсов

Визуально легко обнаружить наличие тупика, но нужны также формальные алгоритмы, реализуемые на компьютере.

Один из таких алгоритмов описан в [Таненбаум, 2002], там же можно найти ссылки на другие алгоритмы.

Существуют и другие способы обнаружения тупиков, применимые также в ситуациях, когда имеется несколько ресурсов каждого типа. Так в [Дейтел, 1987] описан способ, называемый редукцией графа распределения ресурсов, а в [Таненбаум, 2002] – матричный алгоритм.

Восстановление после тупиков

Обнаружив тупик, можно вывести из него систему, нарушив одно из условий существования тупика. При этом, возможно, несколько процессов частично или полностью потеряют результаты проделанной работы.

Сложность восстановления обусловлена рядом факторов:

- В большинстве систем нет достаточно эффективных средств, чтобы приостановить процесс, вывести его из системы и возобновить впоследствии с того места, где он был остановлен.

- Если даже такие средства есть, то их использование требует затрат и внимания оператора.
- Восстановление после тупика может потребовать значительных усилий.

Самый простой и наиболее распространенный способ устранить тупик – завершить выполнение одного или более процессов, чтобы впоследствии использовать его ресурсы. Тогда в случае удачи остальные процессы смогут выполняться. Если это не помогает, можно ликвидировать еще несколько процессов. После каждой ликвидации должен запускаться алгоритм обнаружения тупика.

По возможности лучше ликвидировать тот процесс, который может быть без ущерба возвращен к началу (такие процессы называются *идемпотентными*). Примером такого процесса может служить компиляция. С другой стороны, процесс, который изменяет содержимое базы данных, не всегда может быть корректно запущен повторно.

В некоторых случаях можно временно забрать ресурс у текущего владельца и передать его другому процессу. Возможность забрать ресурс у процесса, дать его другому процессу и затем без ущерба вернуть назад сильно зависит от природы ресурса. Подобное восстановление часто затруднительно, если не невозможно.

В ряде систем реализованы средства отката и перезапуска или рестарта с контрольной точки (сохранение состояния системы в какой-то момент времени). Если проектировщики системы знают, что тупик вероятен, они могут периодически организовывать для процессов контрольные точки. Иногда это приходится делать разработчикам прикладных программ.

Когда тупик обнаружен, видно, какие ресурсы вовлечены в цикл кругового ожидания. Чтобы осуществить восстановление, процесс, который владеет таким ресурсом, должен быть отброшен к моменту времени, предшествующему его запросу на этот ресурс.

Заключение

Возникновение тупиков является потенциальной проблемой любой операционной системы. Они возникают, когда имеется группа процессов, каждый из которых пытается получить исключительный доступ к некоторым ресурсам и претендует на ресурсы, принадлежащие другому процессу. В итоге все они оказываются в состоянии бесконечного ожидания.

С тупиками можно бороться, можно их обнаруживать, избегать и восстанавливать систему после тупиков. Однако цена подобных действий высока и соответствующие усилия должны предприниматься только в системах, где игнорирование тупиковых ситуаций приводит к катастрофическим последствиям.

Часть III. Управление памятью

В данной части рассматривается идеология построения системы управления памятью в современных ОС. Центральная концепция управления памятью – система виртуальной памяти – обеспечивает поддержку и защиту больших виртуальных адресных пространств процессов, составленных из нескольких логических сегментов. Тщательное проектирование аппаратно-зависимых и аппаратно-независимых компонентов менеджера памяти, базирующееся на анализе поведения программ (локальности ссылок), дает возможность организовать их производительную работу.

Лекция 8. Организация памяти компьютера. Простейшие схемы управления памятью

В настоящей лекции рассматриваются простейшие способы управления памятью в ОС. Физическая память компьютера имеет иерархическую структуру. Программа представляет собой набор сегментов в логическом адресном пространстве. ОС осуществляет связывание логических и физических адресных пространств.

Ключевые слова: иерархия памяти, оперативная, основная, физическая, логическая память, адресное пространство, связывание адресов, трансляция адреса, принцип локальности, оверлейная структура, фиксированные разделы, динамические разделы, фрагментация внешняя, фрагментация внутренняя, сегменты, страницы.

Введение

Главная задача компьютерной системы – выполнять программы. Программы вместе с данными, к которым они имеют доступ, в процессе выполнения должны (по крайней мере частично) находиться в оперативной памяти. Операционной системе приходится решать задачу распределения памяти между пользовательскими процессами и компонентами ОС. Эта деятельность называется управлением памятью. Таким образом, память (storage, memory) является важнейшим ресурсом, требующим тщательного управления. В недавнем прошлом память была самым дорогим ресурсом.

Часть ОС, которая отвечает за управление памятью, называется *менеджером памяти*.

Физическая организация памяти компьютера

Запоминающие устройства компьютера разделяют, как минимум, на два уровня: *основную* (*главную, оперативную, физическую*) и *вторичную* (*внешнюю*) память.

Основная память представляет собой упорядоченный массив однобайтовых ячеек, каждая из которых имеет свой уникальный адрес (номер). Процессор извлекает команду из основной памяти, декодирует и выполняет ее. Для выполнения команды могут потребоваться обращения еще к нескольким ячейкам основной памяти. Обычно основная память изготавливается с применением полупроводниковых технологий и теряет свое содержимое при отключении питания.

Вторичную память (это главным образом диски) также можно рассматривать как одномерное линейное адресное пространство, состоящее из последовательности байтов. В отличие от оперативной памяти, она является энергонезависимой, имеет существенно большую емкость и используется в качестве расширения основной памяти.

Эту схему можно дополнить еще несколькими промежуточными уровнями, как показано на рис. 8.1. Разновидности памяти могут быть объединены в иерархию по убыванию времени доступа, возрастанию цены и увеличению емкости.

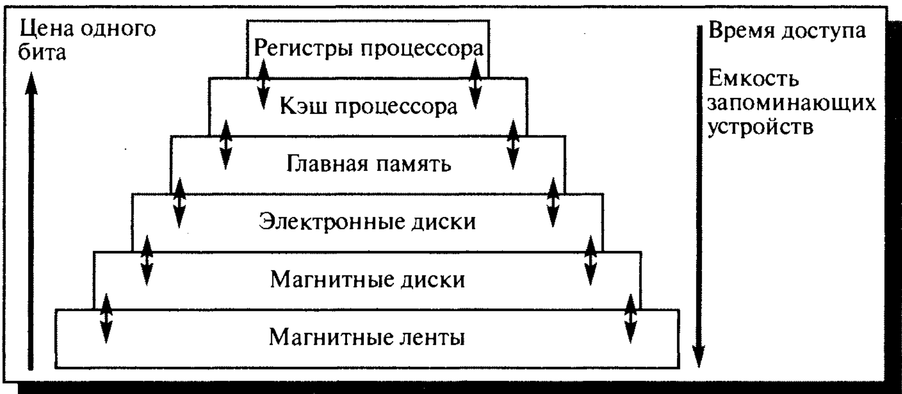


Рис. 8.1. Иерархия памяти

Многоуровневую схему используют следующим образом. Информация, которая находится в памяти верхнего уровня, обычно хранится также на уровнях с большими номерами. Если процессор не обнаруживает нужную информацию на i-м уровне, он начинает искать ее на следующих уровнях. Когда нужная информация найдена, она переносится в более быстрые уровни.

Локальность

Оказывается, при таком способе организации по мере снижения скорости доступа к уровню памяти снижается также и частота обращений к нему.

Ключевую роль здесь играет свойство реальных программ, в течение ограниченного отрезка времени способных работать с небольшим набором адресов памяти. Это эмпирически наблюдаемое свойство известно как *принцип локальности* или *локализации обращений.*

Свойство локальности (соседние в пространстве и времени объекты характеризуются похожими свойствами) присуще не только функционированию ОС, но и природе вообще. В случае ОС свойство локальности объяснимо, если учесть, как пишутся программы и как хранятся данные, то есть обычно в течение какого-то отрезка времени ограниченный фрагмент кода работает с ограниченным набором данных. Эту часть кода и данных удается разместить в памяти с быстрым доступом. В результате реальное время доступа к памяти определяется временем доступа к верхним уровням, что и обусловливает эффективность использования иерархической схемы. Надо сказать, что описываемая организация вычислительной системы во многом имитирует деятельность человеческого мозга при переработке информации. Действительно, решая конкретную проблему, человек работает с небольшим объемом информации, храня не относящиеся к делу сведения в своей памяти или во внешней памяти (например, в книгах).

Кэш процессора обычно является частью аппаратуры, поэтому менеджер памяти ОС занимается распределением информации главным образом в основной и внешней памяти компьютера. В некоторых схемах потоки между оперативной и внешней памятью регулируются программистом (см., например, далее оверлейные структуры), однако это связано с затратами времени программиста, так что подобную деятельность стараются возложить на ОС.

Адреса в основной памяти, характеризующие реальное расположение данных в физической памяти, называются *физическими* адресами. Набор физических адресов, с которым работает программа, называют *физическим адресным пространством.*

Логическая память

Аппаратная организация памяти в виде линейного набора ячеек не соответствует представлениям программиста о том, как организовано хранение программ и данных. Большинство программ представляет собой набор модулей, созданных независимо друг от друга. Иногда все модули,

входящие в состав процесса, располагаются в памяти один за другим, образуя линейное пространство адресов. Однако чаще модули помещаются в разные области памяти и используются по-разному.

Схема управления памятью, поддерживающая этот взгляд пользователя на то, как хранятся программы и данные, называется *сегментацией*. Сегмент – область памяти определенного назначения, внутри которой поддерживается линейная адресация. Сегменты содержат процедуры, массивы, стек или скалярные величины, но обычно не содержат информацию смешанного типа.

По-видимому, вначале сегменты памяти появились в связи с необходимостью обобществления процессами фрагментов программного кода (текстовый редактор, тригонометрические библиотеки и т. д.), без чего каждый процесс должен был хранить в своем адресном пространстве дублирующую информацию. Эти отдельные участки памяти, хранящие информацию, которую система отображает в память нескольких процессов, получили название сегментов. Память, таким образом, перестала быть линейной и превратилась в двумерную. Адрес состоит из двух компонентов: номер сегмента, смещение внутри сегмента. Далее оказалось удобным размещать в разных сегментах различные компоненты процесса (код программы, данные, стек и т. д.). Попутно выяснилось, что можно контролировать характер работы с конкретным сегментом, приписав ему атрибуты, например, права доступа или типы операций, которые разрешается производить с данными, хранящимися в сегменте.

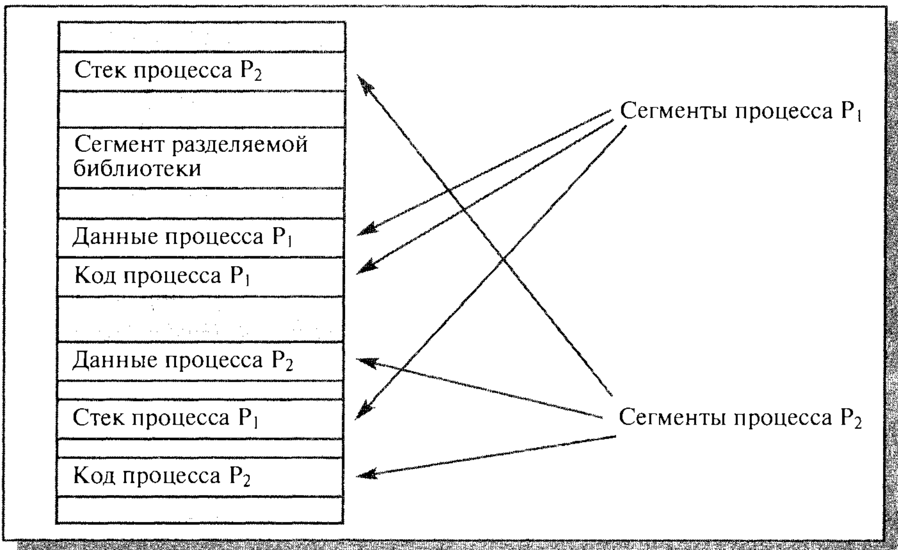


Рис. 8.2. Расположение сегментов процессов в памяти компьютера

Некоторые сегменты, описывающие адресное пространство процесса, показаны на рис. 8.2. Более подробная информация о типах сегментов имеется в лекции 10.

Большинство современных ОС поддерживают сегментную организацию памяти. В некоторых архитектурах (Intel, например) сегментация поддерживается оборудованием.

Адреса, к которым обращается процесс, таким образом, отличаются от адресов, реально существующих в оперативной памяти. В каждом конкретном случае используемые программой адреса могут быть представлены различными способами. Например, адреса в исходных текстах обычно символические. Компилятор связывает эти символические адреса с перемещаемыми адресами (такими, как n байт от начала модуля). Подобный адрес, сгенерированный программой, обычно называют *логическим* (*в системах с виртуальной памятью он часто называется виртуальным*) адресом. Совокупность всех логических адресов называется *логическим* (*виртуальным*) *адресным пространством*.

Связывание адресов

Итак, логические и физические адресные пространства ни по организации, ни по размеру не соответствуют друг другу. Максимальный размер логического адресного пространства обычно определяется разрядностью процессора (например, 2^{32}) и в современных системах значительно превышает размер физического адресного пространства. Следовательно, процессор и ОС должны быть способны отобразить ссылки в коде программы в реальные физические адреса, соответствующие текущему расположению программы в основной памяти. Такое отображение адресов называют *трансляцией* (*привязкой*) *адреса* или *связыванием адресов* (см. рис. 8.3).

Связывание логического адреса, порожденного оператором программы, с физическим должно быть осуществлено до начала выполнения оператора или в момент его выполнения. Таким образом, привязка инструкций и данных к памяти в принципе может быть сделана на следующих шагах [Silberschatz, 2002].

- Этап компиляции (Compile time). Когда на стадии компиляции известно точное место размещения процесса в памяти, тогда непосредственно генерируются физические адреса. При изменении стартового адреса программы необходимо перекомпилировать ее код. В качестве примера можно привести .com программы MS-DOS, которые связывают ее с физическими адресами на стадии компиляции.
- Этап загрузки (Load time). Если информация о размещении программы на стадии компиляции отсутствует, компилятор генерирует перемещаемый код. В этом случае окончательное связывание

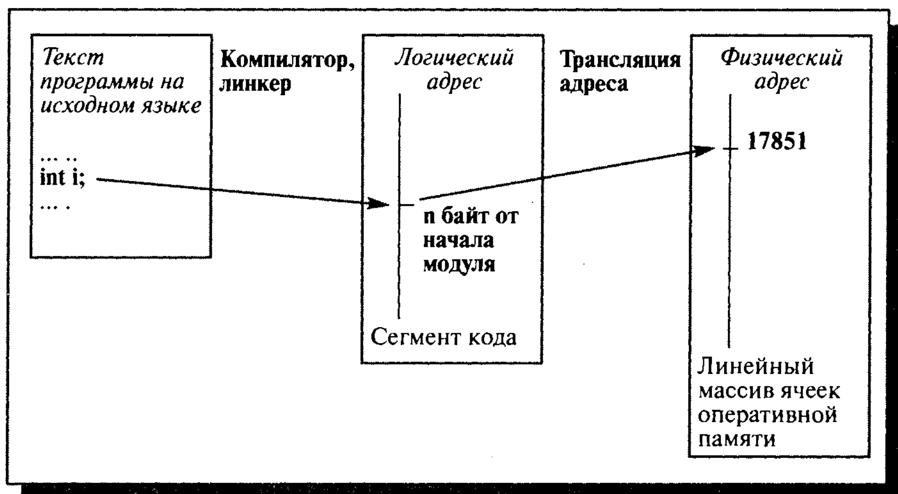


Рис. 8.3. Формирование логического адреса и связывание логического адреса с физическим

откладывается до момента загрузки. Если стартовый адрес меняется, нужно всего лишь перезагрузить код с учетом измененной величины.

- Этап выполнения (Execution time). Если процесс может быть перемещен во время выполнения из одной области памяти в другую, связывание откладывается до стадии выполнения. Здесь желательно наличие специализированного оборудования, например регистров перемещения. Их значение прибавляется к каждому адресу, сгенерированному процессом. Большинство современных ОС осуществляет трансляцию адресов на этапе выполнения, используя для этого специальный аппаратный механизм (см. лекцию 9).

Функции системы управления памятью

Чтобы обеспечить эффективный контроль использования памяти, ОС должна выполнять следующие функции:

- отображение адресного пространства процесса на конкретные области физической памяти;
- распределение памяти между конкурирующими процессами;
- контроль доступа к адресным пространствам процессов;
- выгрузка процессов (целиком или частично) во внешнюю память, когда в оперативной памяти недостаточно места;
- учет свободной и занятой памяти.

В следующих разделах лекции рассматривается ряд конкретных схем управления памятью. Каждая схема включает в себя определенную идео-

логию управления, а также алгоритмы и структуры данных и зависит от архитектурных особенностей используемой системы. Вначале будут рассмотрены простейшие схемы. Доминирующая на сегодня схема виртуальной памяти будет описана в последующих лекциях.

Простейшие схемы управления памятью

Первые ОС применяли очень простые методы управления памятью. Вначале каждый процесс пользователя должен был полностью поместиться в основной памяти, занимать непрерывную область памяти, а система принимала к обслуживанию дополнительные пользовательские процессы до тех пор, пока все они одновременно помещались в основной памяти. Затем появился «простой свопинг» (система по-прежнему размещает каждый процесс в основной памяти целиком, но иногда на основании некоторого критерия целиком сбрасывает образ некоторого процесса из основной памяти во внешнюю и заменяет его в основной памяти образом другого процесса). Такого рода схемы имеют не только историческую ценность. В настоящее время они применяются в учебных и научно-исследовательских модельных ОС, а также в ОС для *встроенных* (embedded) компьютеров.

Схема с фиксированными разделами

Самым простым способом управления оперативной памятью является ее предварительное (обычно на этапе генерации или в момент загрузки системы) разбиение на несколько разделов фиксированной величины. Поступающие процессы помещаются в тот или иной раздел. При этом происходит условное разбиение физического адресного пространства. Связывание логических и физических адресов процесса происходит на этапе его загрузки в конкретный раздел, иногда – на этапе компиляции.

Каждый раздел может иметь свою очередь процессов, а может существовать и глобальная очередь для всех разделов (см. рис. 8.4).

Эта схема была реализована в IBM OS/360 (MFT), DEC RSX-11 и ряде других систем.

Подсистема управления памятью оценивает размер поступившего процесса, выбирает подходящий для него раздел, осуществляет загрузку процесса в этот раздел и настройку адресов.

Очевидный недостаток этой схемы – число одновременно выполняемых процессов ограничено числом разделов.

Другим существенным недостатком является то, что предлагаемая схема сильно страдает от *внутренней фрагментации* – потери части памяти, выделенной процессу, но не используемой им. Фрагментация возникает потому, что процесс не полностью занимает выделенный ему раздел

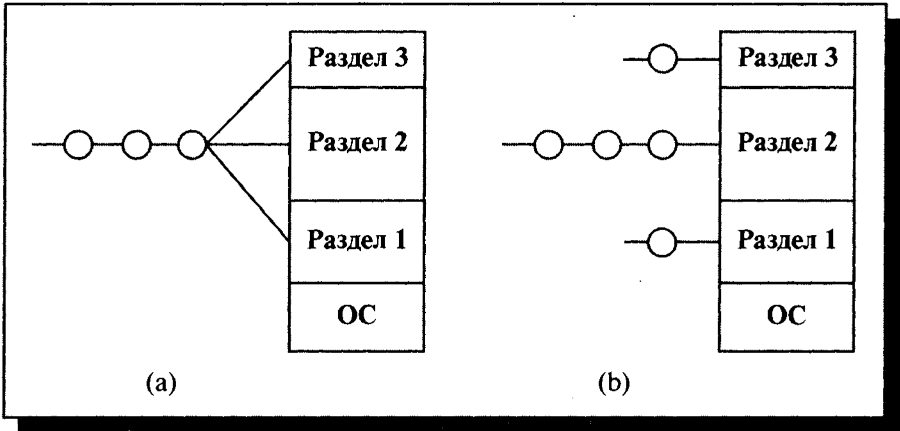


Рис. 8.4. Схема с фиксированными разделами: (a) – с общей очередью процессов, (b) – с отдельными очередями процессов

или потому, что некоторые разделы слишком малы для выполняемых пользовательских программ.

Один процесс в памяти

Частный случай схемы с фиксированными разделами – работа менеджера памяти однозадачной ОС. В памяти размещается один пользовательский процесс. Остается определить, где располагается пользовательская программа по отношению к ОС – в верхней части памяти, в нижней или в средней. Причем часть ОС может быть в ROM (например, BIOS, драйверы устройств). Главный фактор, влияющий на это решение, – расположение вектора прерываний, который обычно локализован в нижней части памяти, поэтому ОС также размещают в нижней. Примером такой организации может служить ОС MS-DOS.

Защита адресного пространства ОС от пользовательской программы может быть организована при помощи одного граничного регистра, содержащего адрес границы ОС.

Оверлейная структура

Так как размер логического адресного пространства процесса может быть больше, чем размер выделенного ему раздела (или больше, чем размер самого большого раздела), иногда используется техника, называемая оверлей (overlay) или организация структуры с перекрытием. Основная

идея – держать в памяти только те инструкции программы, которые нужны в данный момент.

Потребность в таком способе загрузки появляется, если логическое адресное пространство системы мало, например 1 Мбайт (MS-DOS) или даже всего 64 Кбайта (PDP-11), а программа относительно велика. На современных 32-разрядных системах, где виртуальное адресное пространство измеряется гигабайтами, проблемы с нехваткой памяти решаются другими способами (см. раздел «Виртуальная память»).

Коды ветвей оверлейной структуры программы находятся на диске как абсолютные образы памяти и считываются драйвером оверлеев при необходимости. Для описания оверлейной структуры обычно используется специальный несложный язык (overlay description language). Совокупность файлов исполняемой программы дополняется файлом (обычно с расширением `.odl`), описывающим дерево вызовов внутри программы. Для примера, приведенного на рис. 8.5, текст этого файла может выглядеть так:

```
A-(B,C)
C-(D,E)
```

Синтаксис подобного файла может распознаваться загрузчиком. Привязка к физической памяти происходит в момент очередной загрузки одной из ветвей программы.

Оверлеи могут быть полностью реализованы на пользовательском уровне в системах с простой файловой структурой. ОС при этом лишь делает несколько больше операций ввода-вывода. Типовое решение – порождение линкером специальных команд, которые включают загрузчик каждый раз, когда требуется обращение к одной из перекрывающихся ветвей программы.

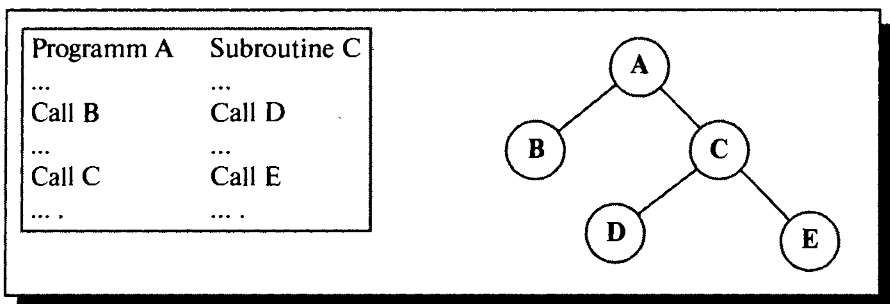


Рис. 8.5. Организация структуры с перекрытием.
Можно поочередно загружать в память ветви A-B, A-C-D и A-C-E программы

Тщательное проектирование оверлейной структуры отнимает много времени и требует знания устройства программы, ее кода, данных и языка описания оверлейной структуры. По этой причине применение оверлеев ограничено компьютерами с небольшим логическим адресным пространством. Как мы увидим в дальнейшем, проблема оверлейных сегментов, контролируемых программистом, отпадает благодаря появлению систем виртуальной памяти.

Заметим, что возможность организации структур с перекрытиями во многом обусловлена свойством локальности, которое позволяет хранить в памяти только ту информацию, которая необходима в конкретный момент вычислений.

Динамическое распределение. Свопинг

Имея дело с пакетными системами, можно обходиться фиксированными разделами и не использовать ничего более сложного. В системах с разделением времени возможна ситуация, когда память не в состоянии содержать все пользовательские процессы. Приходится прибегать к свопингу (swapping) – перемещению процессов из главной памяти на диск и обратно целиком. Частичная выгрузка процессов на диск осуществляется в системах со страничной организацией (paging) и будет рассмотрена ниже.

Выгруженный процесс может быть возвращен в то же самое адресное пространство или в другое. Это ограничение диктуется методом связывания. Для схемы связывания на этапе выполнения можно загрузить процесс в другое место памяти.

Свопинг не имеет непосредственного отношения к управлению памятью, скорее он связан с подсистемой планирования процессов. Очевидно, что свопинг увеличивает время переключения контекста. Время выгрузки может быть сокращено за счет организации специально отведенного пространства на диске (раздел для свопинга). Обмен с диском при этом осуществляется блоками большего размера, то есть быстрее, чем через стандартную файловую систему. Во многих версиях Unix свопинг начинает работать только тогда, когда возникает необходимость в снижении загрузки системы.

Схема с переменными разделами

В принципе, система свопинга может базироваться на фиксированных разделах. Более эффективной, однако, представляется схема динамического распределения или схема с переменными разделами, которая может использоваться и в тех случаях, когда все процессы целиком поме-

щаются в памяти, то есть в отсутствие свопинга. В этом случае вначале вся память свободна и не разделена заранее на разделы. Вновь поступающей задаче выделяется строго необходимое количество памяти, не более. После выгрузки процесса память временно освобождается. По истечении некоторого времени память представляет собой переменное число разделов разного размера (рис. 8.6). Смежные свободные участки могут быть объединены.

В какой раздел помещать процесс? Наиболее распространены три стратегии:

- Стратегия первого подходящего (First fit). Процесс помещается в первый подходящий по размеру раздел.
- Стратегия наиболее подходящего (Best fit). Процесс помещается в тот раздел, где после его загрузки останется меньше всего свободного места.
- Стратегия наименее подходящего (Worst fit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

Моделирование показало, что доля полезно используемой памяти в первых двух случаях больше, при этом первый способ несколько быстрее. Попутно заметим, что перечисленные стратегии широко применяются и другими компонентами ОС, например для размещения файлов на диске.

Типовой цикл работы менеджера памяти состоит в анализе запроса на выделение свободного участка (раздела), выборе его среди имеющихся в соответствии с одной из стратегий (первого подходящего, наиболее подходящего и наименее подходящего), загрузке процесса в выбранный раздел и последующих изменениях таблиц свободных и занятых областей. Аналогичная корректировка необходима и после завершения процесса. Связывание адресов может осуществляться на этапах загрузки и выполнения.

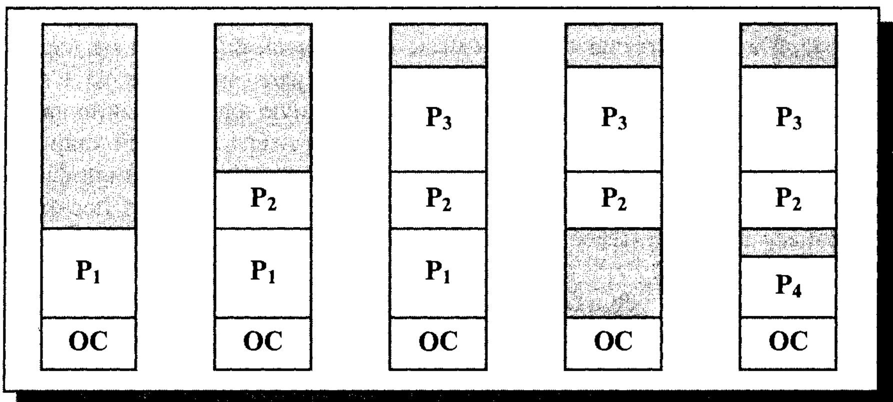


Рис. 8.6. Динамика распределения памяти между процессами (серым цветом показана неиспользуемая память)

Этот метод более гибок по сравнению с методом фиксированных разделов, однако ему присуща *внешняя фрагментация* – наличие большого числа участков неиспользуемой памяти, не выделенной ни одному процессу. Выбор стратегии размещения процесса между первым подходящим и наиболее подходящим слабо влияет на величину фрагментации. Любопытно, что метод наиболее подходящего может оказаться наихудшим, так как он оставляет множество мелких незанятых блоков.

Статистический анализ показывает, что пропадает в среднем 1/3 памяти! Это известное правило 50% (два соседних свободных участка в отличие от двух соседних процессов могут быть объединены).

Одно из решений проблемы внешней фрагментации – организовать *сжатие*, то есть перемещение всех занятых (свободных) участков в сторону возрастания (убывания) адресов, так, чтобы вся свободная память образовала непрерывную область. Этот метод иногда называют схемой с *перемещаемыми разделами*. В идеале фрагментация после сжатия должна отсутствовать. Сжатие, однако, является дорогостоящей процедурой, алгоритм выбора оптимальной стратегии сжатия очень труден и, как правило, сжатие осуществляется в комбинации с выгрузкой и загрузкой по другим адресам.

Страничная память

Описанные выше схемы недостаточно эффективно используют память, поэтому в современных схемах управления памятью не принято размещать процесс в оперативной памяти одним непрерывным блоком.

В самом простом и наиболее распространенном случае страничной организации памяти (или paging) как логическое адресное пространство, так и физическое представляются состоящими из наборов блоков или страниц одинакового размера. При этом образуются логические страницы (page), а соответствующие единицы в физической памяти называют физическими страницами или страничными кадрами (page frames). Страницы (и страничные кадры) имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться. Каждый кадр содержит одну страницу данных. При такой организации внешняя фрагментация отсутствует, а потери из-за внутренней фрагментации, поскольку процесс занимает целое число страниц, ограничены частью последней страницы процесса.

Логический адрес в страничной системе – упорядоченная пара (p, d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, на которой размещается адресуемый элемент. Заметим, что разбиение адресного пространства на страницы осуществляется вычислительной системой незаметно для программиста. Поэтому адрес является двумерным лишь с точки зрения операционной системы, а с

точки зрения программиста адресное пространство процесса остается линейным.

Описываемая схема позволяет загрузить процесс, даже если нет непрерывной области кадров, достаточной для размещения процесса целиком. Но одного базового регистра для осуществления трансляции адреса в данной схеме недостаточно. Система отображения логических адресов в физические сводится к системе отображения логических страниц в физические и представляет собой *таблицу страниц*, которая хранится в оперативной памяти. Иногда говорят, что таблица страниц – это кусочно-линейная функция отображения, заданная в табличном виде.

Интерпретация логического адреса показана на рис. 8.7. Если выполняемый процесс обращается к логическому адресу v = (p, d), механизм отображения ищет номер страницы p в таблице страниц и определяет, что эта страница находится в страничном кадре p', формируя реальный адрес из p' и d.

Таблица страниц (page table) адресуется при помощи специального регистра процессора и позволяет определить номер кадра по логическому адресу. Помимо этой основной задачи, при помощи атрибутов, записанных в строке таблицы страниц, можно организовать контроль доступа к конкретной странице и ее защиту.

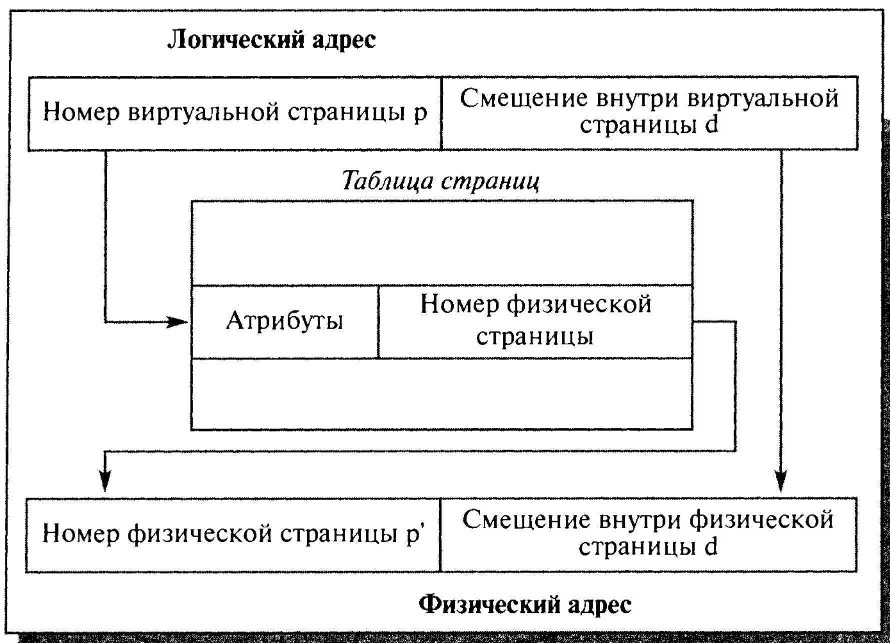


Рис. 8.7. Связь логического и физического адресов при страничной организации памяти

Отметим еще раз различие точек зрения пользователя и системы на используемую память. С точки зрения пользователя, его память – единое непрерывное пространство, содержащее только одну программу. Реальное отображение скрыто от пользователя и контролируется ОС. Заметим, что процессу пользователя чужая память недоступна. Он не имеет возможности адресовать память за пределами своей таблицы страниц, которая включает только его собственные страницы.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающий его состояние.

Отображение адресов должно быть осуществлено корректно даже в сложных случаях и обычно реализуется аппаратно. Для ссылки на таблицу процессов используется специальный регистр. При переключении процессов необходимо найти таблицу страниц нового процесса, указатель на которую входит в контекст процесса.

Сегментная и сегментно-страничная организация памяти

Существуют две другие схемы организации управления памятью: сегментная и сегментно-страничная. Сегменты, в отличие от страниц, могут иметь переменный размер. Идея сегментации изложена во введении. При сегментной организации виртуальный адрес является двумерным как для программиста, так и для операционной системы, и состоит из двух полей – номера сегмента и смещения внутри сегмента. Подчеркнем, **что в отличие от страничной организации, где линейный адрес преобразован в двумерный операционной системой для удобства отображения, здесь двумерность адреса является следствием представления пользователя о процессе не в виде линейного массива байтов, а как набора сегментов переменного размера (данные, код, стек...).**

Программисты, пишущие на языках низкого уровня, должны иметь представление о сегментной организации, явным образом меняя значения сегментных регистров (это хорошо видно по текстам программ, написанных на Ассемблере). Логическое адресное пространство – набор сегментов. Каждый сегмент имеет имя, размер и другие параметры (уровень привилегий, разрешенные виды обращений, флаги присутствия...). В отличие от страничной схемы, где пользователь задает только один адрес, который разбивается на номер страницы и смещение прозрачным для программиста образом, в сегментной схеме пользователь специфицирует каждый адрес двумя величинами: именем сегмента и смещением.

Каждый сегмент – линейная последовательность адресов, начинающаяся с 0. Максимальный размер сегмента определяется разрядностью

процессора (при 32-разрядной адресации это 2^{32} байт или 4 Гбайт). Размер сегмента может меняться динамически (например, сегмент стека). В элементе таблицы сегментов помимо физического адреса начала сегмента обычно содержится и длина сегмента. Если размер смещения в виртуальном адресе выходит за пределы размера сегмента, возникает исключительная ситуация.

Логический адрес – упорядоченная пара v = (s, d), номер сегмента и смещение внутри сегмента.

В системах, где сегменты поддерживаются аппаратно, эти параметры обычно хранятся в таблице дескрипторов сегментов, а программа обращается к этим дескрипторам по номерам-селекторам. При этом в контекст каждого процесса входит набор сегментных регистров, содержащих селекторы текущих сегментов кода, стека, данных и т. д. и определяющих, какие сегменты будут использоваться при разных видах обращений к памяти. Это позволяет процессору уже на аппаратном уровне определять допустимость обращений к памяти, упрощая реализацию защиты информации от повреждения и несанкционированного доступа.

Аппаратная поддержка сегментов распространена мало (главным образом на процессорах Intel). В большинстве ОС сегментация реализуется на уровне, не зависящем от аппаратуры.

Хранить в памяти сегменты большого размера целиком так же неудобно, как и хранить процесс непрерывным блоком. Напрашивается идея разбиения сегментов на страницы. При *сегментно-страничной* орга-

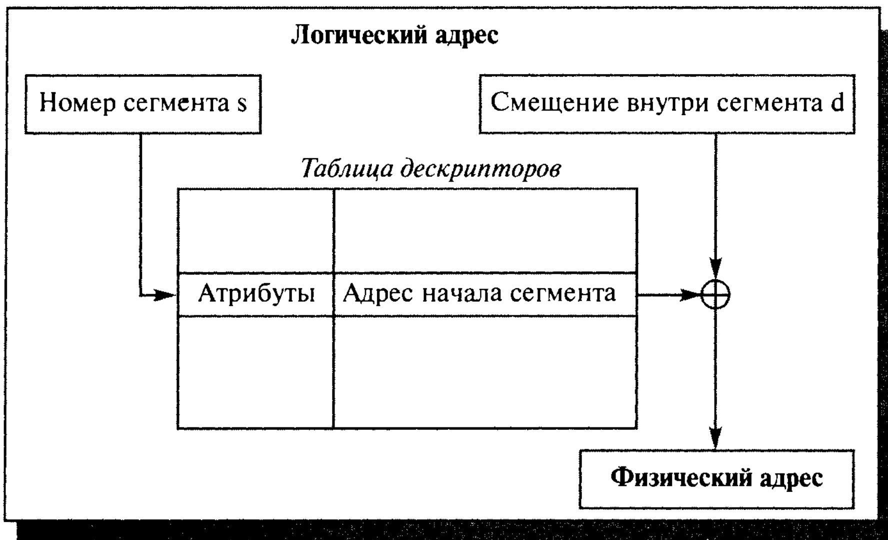


Рис. 8.8. Преобразование логического адреса при сегментной организации памяти

низации памяти происходит двухуровневая трансляция виртуального адреса в физический. В этом случае логический адрес состоит из трех полей: номера сегмента логической памяти, номера страницы внутри сегмента и смещения внутри страницы. Соответственно, используются две таблицы отображения – таблица сегментов, связывающая номер сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента.

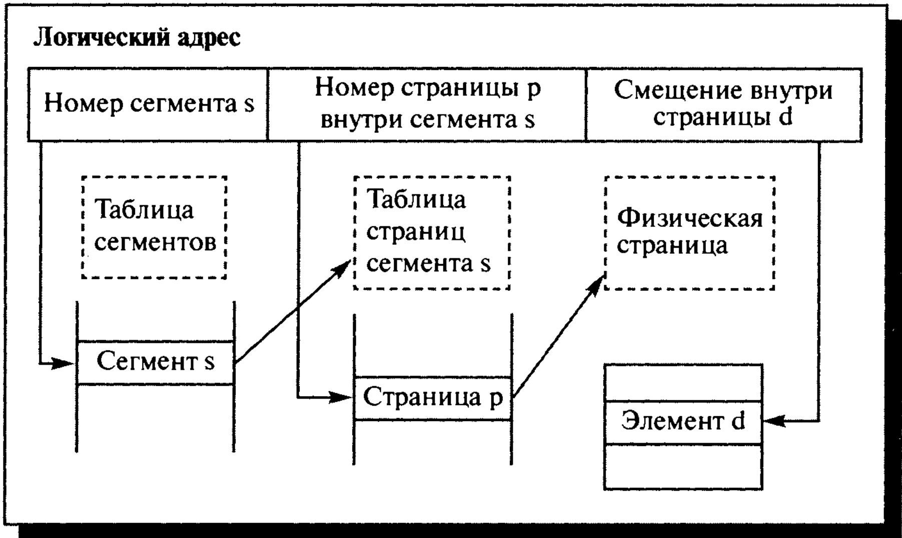


Рис. 8.9. Упрощенная схема формирования физического адреса при сегментно-страничной организации памяти

Сегментно-страничная и страничная организация памяти позволяют легко организовать совместное использование одних и тех же данных и программного кода разными задачами. Для этого различные логические блоки памяти разных процессов отображают в один и тот же блок физической памяти, где размещается разделяемый фрагмент кода или данных.

Заключение

В настоящей лекции описаны простейшие способы управления памятью в ОС. Физическая память компьютера имеет иерархическую структуру. Программа представляет собой набор сегментов в логическом адресном пространстве. ОС осуществляет связывание логических и физических адресных пространств. В последующих лекциях будут рассматриваться современные решения, связанные с поддержкой виртуальной памяти.

Лекция 9. Виртуальная память. Архитектурные средства поддержки виртуальной памяти

Рассмотрены аппаратные особенности поддержки виртуальной памяти. Разбиение адресного пространства процесса на части и динамическая трансляция адреса позволили выполнять процесс даже в отсутствие некоторых его компонентов в оперативной памяти. Следствием такой стратегии является возможность выполнения больших программ, размер которых может превышать размер оперативной памяти.

Ключевые слова: виртуальная память, таблица страниц, бит присутствия, ссылки, модификации, виртуальный адрес, страничная организация, сегментно-страничная организация, многоуровневая таблица страниц, ассоциативная память, инвертированная таблица страниц.

В этой и следующей лекциях речь пойдет о наиболее распространенной в настоящее время схеме управления памятью, известной как виртуальная память, в рамках которой осуществляется сложная связь между аппаратным и программным обеспечением. Вначале будут рассмотрены аппаратные аспекты виртуальной памяти, а затем вопросы, возникающие при ее программной реализации.

Понятие виртуальной памяти

Разработчикам программного обеспечения часто приходится решать проблему размещения в памяти больших программ, размер которых превышает объем доступной оперативной памяти. Один из вариантов решения данной проблемы – организация структур с перекрытием – рассмотрен в предыдущей лекции. При этом предполагалось активное участие программиста в процессе формирования перекрывающихся частей программы. Развитие архитектуры компьютеров и расширение возможностей операционной системы по управлению памятью позволило переложить решение этой задачи на компьютер. Одним из главных достижений стало появление виртуальной памяти (virtual memory). Впервые она была реализована в 1959 году на компьютере «Атлас», разработанном в Манчестерском университете.

Суть концепции виртуальной памяти заключается в следующем. Информация, с которой работает активный процесс, должна располагаться в оперативной памяти. В схемах виртуальной памяти у процесса создается иллюзия того, что вся необходимая ему информация имеется в основной

памяти. Для этого, во-первых, занимаемая процессом память разбивается на несколько частей, например страниц. Во-вторых, логический адрес (логическая страница), к которому обращается процесс, динамически транслируется в физический адрес (физическую страницу). И наконец, в тех случаях, когда страница, к которой обращается процесс, не находится в физической памяти, нужно организовать ее подкачку с диска. Для контроля наличия страницы в памяти вводится специальный бит присутствия, входящий в состав атрибутов страницы в таблице страниц.

Таким образом, в наличии всех компонентов процесса в основной памяти необходимости нет. Важным следствием такой организации является то, что размер памяти, занимаемой процессом, может быть больше, чем размер оперативной памяти. Принцип локальности обеспечивает этой схеме нужную эффективность.

Возможность выполнения программы, находящейся в памяти лишь частично, имеет ряд вполне очевидных преимуществ:

- Программа не ограничена объемом физической памяти. Упрощается разработка программ, поскольку можно задействовать большие виртуальные пространства, не заботясь о размере используемой памяти.
- Поскольку появляется возможность частичного помещения программы (процесса) в память и гибкого перераспределения памяти между программами, можно разместить в памяти больше программ, что увеличивает загрузку процессора и пропускную способность системы.
- Объем ввода-вывода для выгрузки части программы на диск может быть меньше, чем в варианте классического свопинга, в итоге каждая программа будет работать быстрее.

Таким образом, возможность обеспечения (при поддержке операционной системы) для программы «видимости» практически неограниченной (характерный размер для 32-разрядных архитектур 2^{32} = 4 Гбайт) адресуемой пользовательской памяти (логическое адресное пространство) при наличии основной памяти существенно меньших размеров (физическое адресное пространство) – очень важный аспект.

Но введение виртуальной памяти позволяет решать другую, не менее важную задачу – обеспечение контроля доступа к отдельным сегментам памяти и, в частности, *защиту* пользовательских программ друг от друга и защиту ОС от пользовательских программ. Каждый процесс работает со своими виртуальными адресами, трансляцию которых в физические выполняет аппаратура компьютера. Таким образом, пользовательский процесс лишен возможности напрямую обратиться к страницам основной памяти, занятым информацией, относящейся к другим процессам.

Например, 16-разрядный компьютер PDP-11/70 с 64 Кбайт логической памяти мог иметь до 2 Мбайт оперативной памяти. Операционная

система этого компьютера тем не менее поддерживала виртуальную память, которая *обеспечивала защиту* и перераспределение основной памяти между пользовательскими процессами.

Напомним, что в системах с виртуальной памятью те адреса, которые генерирует программа (логические адреса), называются виртуальными, и они формируют виртуальное адресное пространство. Термин «виртуальная память» означает, что программист имеет дело с памятью, отличной от реальной, размер которой потенциально больше, чем размер оперативной памяти.

Хотя известны и чисто программные реализации виртуальной памяти, это направление получило наиболее широкое развитие после соответствующей аппаратной поддержки.

Следует отметить, что оборудование компьютера принимает участие в трансляции адреса практически во всех схемах управления памятью. Но в случае виртуальной памяти это становится более сложным вследствие разрывности отображения и многомерности логического адресного пространства. Может быть, наиболее существенным вкладом аппаратуры в реализацию описываемой схемы является автоматическая генерация исключительных ситуаций при отсутствии в памяти нужных страниц (page fault).

Любая из трех ранее рассмотренных схем управления памятью – страничной, сегментной и сегментно-страничной – пригодна для организации виртуальной памяти. Чаще всего используется сегментно-страничная модель, которая является синтезом страничной модели и идеи сегментации. Причем для тех архитектур, в которых сегменты не поддерживаются аппаратно, их реализация – задача архитектурно-независимого компонента менеджера памяти.

Сегментная организация в чистом виде встречается редко.

Архитектурные средства поддержки виртуальной памяти

Очевидно, что невозможно создать полностью машинно-независимый компонент управления виртуальной памятью. С другой стороны, имеются существенные части программного обеспечения, связанного с управлением виртуальной памятью, для которых детали аппаратной реализации совершенно не важны. Одним из достижений современных ОС является грамотное и эффективное разделение средств управления виртуальной памятью на аппаратно-независимую и аппаратно-зависимую части. Коротко рассмотрим, что и каким образом входит в аппаратно-зависимую часть подсистемы управления виртуальной памятью. Компоненты

аппаратно-независимой подсистемы будут рассмотрены в следующей лекции.

В самом распространенном случае необходимо отобразить большое виртуальное адресное пространство в физическое адресное пространство существенно меньшего размера. Пользовательский процесс или ОС должны иметь возможность осуществить запись по виртуальному адресу, а задача ОС – сделать так, чтобы записанная информация оказалась в физической памяти (впоследствии при нехватке оперативной памяти она может быть вытеснена во внешнюю память). В случае виртуальной памяти система отображения адресных пространств помимо трансляции адресов должна предусматривать ведение таблиц, показывающих, какие области виртуальной памяти в данный момент находятся в физической памяти и где именно размещаются.

Страничная виртуальная память

Как и в случае простой страничной организации, страничная виртуальная память и физическая память представляются состоящими из наборов блоков или страниц одинакового размера. Виртуальные адреса делятся на страницы (page), соответствующие единицы в физической памяти образуют страничные кадры (page frames), а в целом система поддержки страничной виртуальной памяти называется пейджингом (paging). Передача информации между памятью и диском всегда осуществляется целыми страницами.

После разбиения менеджером памяти виртуального адресного пространства на страницы виртуальный адрес преобразуется в упорядоченную пару (p, d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, внутри которой размещается адресуемый элемент. Процесс может выполняться, если его текущая страница находится в оперативной памяти. Если текущей страницы в главной памяти нет, она должна быть переписана (подкачана) из внешней памяти. Поступившую страницу можно поместить в любой свободный страничный кадр.

Поскольку число виртуальных страниц велико, таблица страниц принимает специфический вид (**см. раздел «Структура таблицы страниц»**), структура записей становится более сложной, среди атрибутов страницы появляются биты присутствия, модификации и другие управляющие биты.

При отсутствии страницы в памяти в процессе выполнения команды возникает *исключительная ситуация*, называемая *страничное нарушение* (*page fault*) или страничный отказ. Обработка страничного нарушения заключается в том, что выполнение команды прерывается, затребованная страница подкачивается из конкретного места вторичной памяти

в свободный страничный кадр физической памяти и попытка выполнения команды повторяется. При отсутствии свободных страничных кадров на диск выгружается редко используемая страница. Проблемы замещения страниц и обработки страничных нарушений рассматриваются в следующей лекции.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающий его состояние.

В большинстве современных компьютеров со страничной организацией в основной памяти хранится лишь часть таблицы страниц, а быстрота доступа к элементам таблицы текущей виртуальной памяти достигается, как будет показано ниже, за счет использования сверхбыстродействующей памяти, размещенной в кэше процессора.

Сегментно-страничная организации виртуальной памяти

Как и в случае простой сегментации, в схемах виртуальной памяти сегмент – это линейная последовательность адресов, начинающаяся с 0. При организации виртуальной памяти размер сегмента может быть велик, например может превышать размер оперативной памяти. Повторяя все ранее приведенные рассуждения о размещении в памяти больших программ, приходим к разбиению сегментов на страницы и необходимости поддержки своей таблицы страниц для каждого сегмента.

На практике, однако, появления в системе большого количества таблиц страниц стараются избежать, организуя неперекрывающиеся сегменты в одном виртуальном пространстве, для описания которого хватает одной таблицы страниц. Таким образом, одна таблица страниц отводится для всего процесса. Например, в популярных ОС Linux и Windows 2000 все сегменты процесса, а также область памяти ядра ограничены виртуальным адресным пространством объемом 4 Гбайт. При этом ядро ОС располагается по фиксированным виртуальным адресам вне зависимости от выполняемого процесса.

Структура таблицы страниц

Организация таблицы страниц – один из ключевых элементов отображения адресов в страничной и сегментно-страничной схемах. Рассмотрим структуру таблицы страниц для случая страничной организации более подробно.

Итак, виртуальный адрес состоит из виртуального номера страницы и смещения. Номер записи в таблице страниц соответствует номеру виртуальной страницы. Размер записи колеблется от системы к системе, но

чаще всего он составляет 32 бита. Из этой записи в таблице страниц находится номер кадра для данной виртуальной страницы, затем прибавляется смещение и формируется физический адрес. Помимо этого запись в таблице страниц содержит информацию об атрибутах страницы. Это биты присутствия и защиты (например, 0 – read/write, 1 – read only...). Также могут быть указаны: бит модификации, который устанавливается, если содержимое страницы модифицировано, и позволяет контролировать необходимость перезаписи страницы на диск; бит ссылки, который помогает выделить малоиспользуемые страницы; бит, разрешающий кэширование, и другие управляющие биты. Заметим, что адреса страниц на диске не являются частью таблицы страниц.

Основную проблему для эффективной реализации таблицы страниц создают большие размеры виртуальных адресных пространств современных компьютеров, которые обычно определяются разрядностью архитектуры процессора. Самыми распространенными на сегодня являются 32-разрядные процессоры, позволяющие создавать виртуальные адресные пространства размером 4 Гбайт (для 64-разрядных компьютеров эта величина равна 2^{64} байт). Кроме того, существует проблема скорости отображения, которая решается за счет использования так называемой ассоциативной памяти (см. следующий раздел).

Подсчитаем примерный размер таблицы страниц. В 32-битном адресном пространстве при размере страницы 4 Кбайт (Intel) получаем $2^{32}/2^{12} = 2^{20}$, то есть приблизительно миллион страниц, а в 64-битном и того более. Таким образом, таблица должна иметь примерно миллион строк (entry), причем запись в строке состоит из нескольких байтов. Заметим, что каждый процесс нуждается в своей таблице страниц (а в случае сегментно-страничной схемы желательно иметь по одной таблице страниц на каждый сегмент).

Понятно, что количество памяти, отводимое таблицам страниц, не может быть так велико. Для того чтобы избежать размещения в памяти огромной таблицы, ее разбивают на ряд фрагментов. В оперативной памяти хранят лишь некоторые, необходимые для конкретного момента исполнения фрагменты таблицы страниц. В силу свойства локальности число таких фрагментов относительно невелико. Выполнить разбиение таблицы страниц на части можно по-разному. Наиболее распространенный способ разбиения – организация так называемой многоуровневой таблицы страниц. Для примера рассмотрим двухуровневую таблицу с размером страниц 4 Кбайт, реализованную в 32-разрядной архитектуре Intel.

Таблица, состоящая из 2^{20} строк, разбивается на 2^{10} таблиц второго уровня по 2^{10} строк. Эти таблицы второго уровня объединены в общую структуру при помощи одной таблицы первого уровня, состоящей из 2^{10} строк. 32-разрядный адрес делится на 10-разрядное поле p_1, 10-разрядное

поле p_2 и 12-разрядное смещение d. Поле p_1 указывает на нужную строку в таблице первого уровня, поле p_2 – второго, а поле d локализует нужный байт внутри указанного страничного кадра (см. рис. 9.1).

При помощи всего лишь одной таблицы второго уровня можно охватить 4 Мбайт (4 Кбайт × 1024) оперативной памяти. Таким образом, для размещения процесса с большим объемом занимаемой памяти достаточно иметь в оперативной памяти одну таблицу первого уровня и несколько таблиц второго уровня. Очевидно, что суммарное количество строк в этих таблицах много меньше 2^{20}. Такой подход естественным образом обобщается на три и более уровней таблицы.

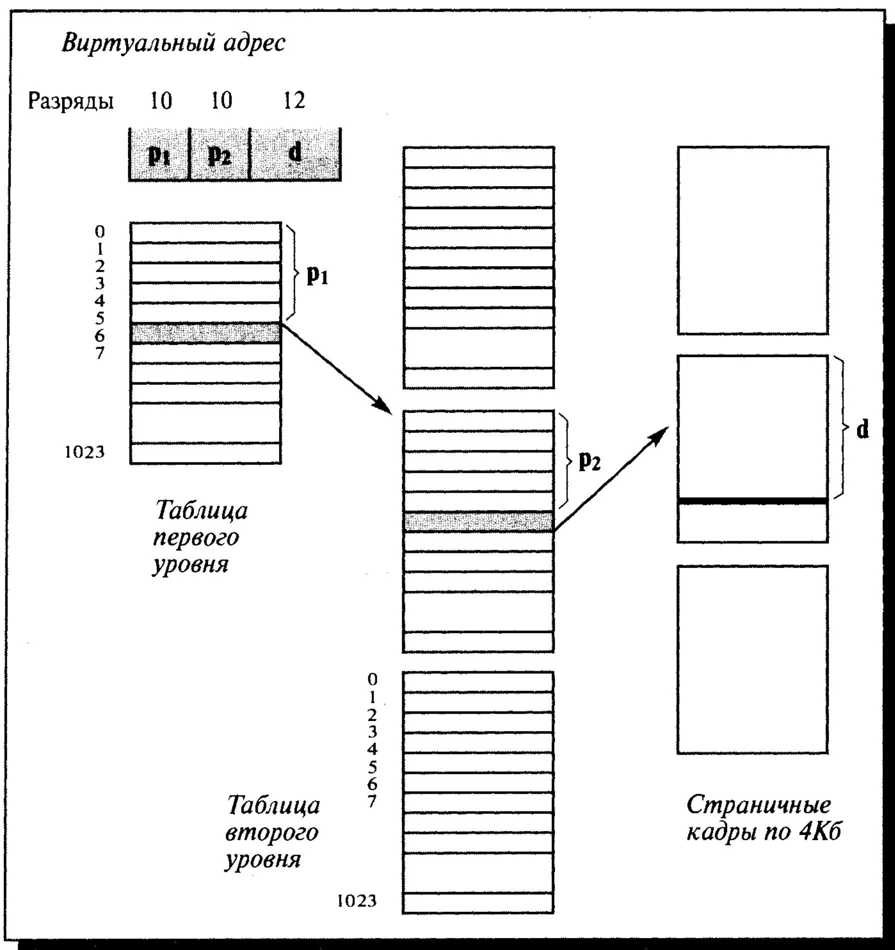


Рис. 9.1. Пример двухуровневой таблицы страниц

Наличие нескольких уровней, естественно, снижает производительность менеджера памяти. Несмотря на то что размеры таблиц на каждом уровне подобраны так, чтобы таблица помещалась целиком внутри одной страницы, обращение к каждому уровню – это отдельное обращение к памяти. Таким образом, трансляция адреса может потребовать нескольких обращений к памяти.

Количество уровней в таблице страниц зависит от конкретных особенностей архитектуры. Можно привести примеры реализации одноуровневого (DEC PDP-11), двухуровневого (Intel, DEC VAX), трехуровневого (Sun SPARC, DEC Alpha) пейджинга, а также пейджинга с заданным количеством уровней (Motorola). Функционирование RISC-процессора MIPS R2000 осуществляется вообще без таблицы страниц. Здесь поиск нужной страницы, если эта страница отсутствует в ассоциативной памяти, должна взять на себя ОС (так называемый zero level paging).

Ассоциативная память

Поиск номера кадра, соответствующего нужной странице, в многоуровневой таблице страниц требует нескольких обращений к основной памяти, поэтому занимает много времени. В некоторых случаях такая задержка недопустима. Проблема ускорения поиска решается на уровне архитектуры компьютера.

В соответствии со свойством локальности большинство программ в течение некоторого промежутка времени обращаются к небольшому количеству страниц, поэтому активно используется только небольшая часть таблицы страниц.

Естественное решение проблемы ускорения – снабдить компьютер аппаратным устройством для отображения виртуальных страниц в физические без обращения к таблице страниц, то есть иметь небольшую, быструю кэш-память, хранящую необходимую на данный момент часть таблицы страниц. Это устройство называется *ассоциативной памятью*, иногда также употребляют термин *буфер поиска трансляции* (translation lookaside buffer – TLB).

Одна запись таблицы в ассоциативной памяти (один вход) содержит информацию об одной виртуальной странице: ее атрибуты и кадр, в котором она находится. Эти поля в точности соответствуют полям в таблице страниц.

Так как ассоциативная память содержит только некоторые из записей таблицы страниц, каждая запись в TLB должна включать поле с номером виртуальной страницы. Память называется ассоциативной, потому что в ней происходит одновременное сравнение номера отображаемой виртуальной страницы с соответствующим полем во всех строках этой

небольшой таблицы. Поэтому данный вид памяти достаточно дорого стоит. В строке, поле виртуальной страницы которой совпало с искомым значением, находится номер страничного кадра. Обычное число записей в TLB от 8 до 4096. Рост количества записей в ассоциативной памяти должен осуществляться с учетом таких факторов, как размер кэша основной памяти и количества обращений к памяти при выполнении одной команды.

Рассмотрим функционирование менеджера памяти при наличии ассоциативной памяти.

Вначале информация об отображении виртуальной страницы в физическую отыскивается в ассоциативной памяти. Если нужная запись найдена – все нормально, за исключением случаев нарушения привилегий, когда запрос на обращение к памяти отклоняется.

Если нужная запись в ассоциативной памяти отсутствует, отображение осуществляется через таблицу страниц. Происходит замена одной из записей в ассоциативной памяти найденной записью из таблицы страниц. Здесь мы сталкиваемся с традиционной для любого кэша проблемой замещения (а именно — какую из записей в кэше необходимо изменить). Конструкция ассоциативной памяти должна организовывать записи таким образом, чтобы можно было принять решение о том, какая из старых записей должна быть удалена при внесении новых.

Число удачных поисков номера страницы в ассоциативной памяти по отношению к общему числу поисков называется hit (совпадение) ratio (пропорция, отношение). Иногда также используется термин «процент попаданий в кэш». Таким образом, hit ratio – часть ссылок, которая может быть сделана с использованием ассоциативной памяти. Обращение к одним и тем же страницам повышает hit ratio. Чем больше hit ratio, тем меньше среднее время доступа к данным, находящимся в оперативной памяти.

Предположим, например, что для определения адреса в случае кэш-промаха через таблицу страниц необходимо 100 нс, а для определения адреса в случае кэш-попадания через ассоциативную память – 20 нс. С 90% hit ratio среднее время определения адреса – $0{,}9 \times 20 + 0{,}1 \times 100 = 28$ нс.

Вполне приемлемая производительность современных ОС доказывает эффективность использования ассоциативной памяти. Высокое значение вероятности нахождения данных в ассоциативной памяти связано с наличием у данных объективных свойств: пространственной и временной локальности.

Необходимо обратить внимание на следующий факт. При переключении контекста процессов нужно добиться того, чтобы новый процесс «не видел» в ассоциативной памяти информацию, относящуюся к преды-

дущему процессу, например очищать ее. Таким образом, использование ассоциативной памяти увеличивает время переключения контекста.

Рассмотренная двухуровневая (ассоциативная память + таблица страниц) схема преобразования адреса является ярким примером иерархии памяти, основанной на использовании принципа локальности, о чем говорилось во введении к предыдущей лекции.

Инвертированная таблица страниц

Несмотря на многоуровневую организацию, хранение нескольких таблиц страниц большого размера по-прежнему представляют собой проблему. Ее значение особенно актуально для 64-разрядных архитектур, где число виртуальных страниц очень велико. Вариантом решения является применение *инвертированной* таблицы страниц (inverted page table). Этот подход применяется на машинах PowerPC, некоторых рабочих станциях Hewlett-Packard, IBM RT, IBM AS/400 и ряде других.

В этой таблице содержится по одной записи на каждый страничный кадр физической памяти. Существенно, что достаточно одной таблицы для всех процессов. Таким образом, для хранения функции отображения требуется фиксированная часть основной памяти, независимо от разрядности архитектуры, размера и количества процессов. Например, для компьютера Pentium с 256 Мбайт оперативной памяти нужна таблица размером 64 Кбайт строк.

Несмотря на экономию оперативной памяти, применение инвертированной таблицы имеет существенный минус – записи в ней (как и в ассоциативной памяти) не отсортированы по возрастанию номеров виртуальных страниц, что усложняет трансляцию адреса. Один из способов решения данной проблемы – использование хеш-таблицы виртуальных адресов. При этом часть виртуального адреса, представляющая собой номер страницы, отображается в хеш-таблицу с использованием функции хеширования. Каждой странице физической памяти здесь соответствует одна запись в хеш-таблице и инвертированной таблице страниц. Виртуальные адреса, имеющие одно значение хеш-функции, сцепляются друг с другом. Обычно длина цепочки не превышает двух записей.

Размер страницы

Разработчики ОС для существующих машин редко имеют возможность влиять на размер страницы. Однако для вновь создаваемых компьютеров решение относительно оптимального размера страницы является актуальным. Как и следовало ожидать, не существует одного наилучшего

размера. Скорее есть набор факторов, влияющих на размер. Обычно размер страницы – это степень двойки от 2^9 до 2^{14} байт.

Чем больше размер страницы, тем меньше будет размер структур данных, обслуживающих преобразование адресов, но тем больше будут потери, связанные с тем, что память можно выделять только постранично.

Как следует выбирать размер страницы? Во-первых, нужно учитывать размер таблицы страниц, здесь желателен большой размер страницы (страниц меньше, соответственно и таблица страниц меньше). С другой стороны, память лучше утилизируется с маленьким размером страницы. В среднем половина последней страницы процесса пропадает. Необходимо также учитывать объем ввода-вывода для взаимодействия с внешней памятью и другие факторы. Проблема не имеет идеального решения. Историческая тенденция состоит в увеличении размера страницы.

Как правило, размер страниц задается аппаратно, например в DEC PDP-11 – 8 Кбайт, в DEC VAX – 512 байт, в других архитектурах, таких как Motorola 68030, размер страниц может быть задан программно. Учитывая все обстоятельства, в ряде архитектур возникают множественные размеры страниц, например в Pentium размер страницы колеблется от 4 Кбайт до 8 Кбайт. Тем не менее большинство коммерческих ОС ввиду сложности перехода на множественный размер страниц поддерживают только один размер страниц.

Заключение

В настоящей лекции рассмотрены аппаратные особенности поддержки виртуальной памяти. Разбиение адресного пространства процесса на части и динамическая трансляция адреса позволили выполнять процесс даже в отсутствие некоторых его компонентов в оперативной памяти. Подкачка недостающих компонентов с диска осуществляется операционной системой в тот момент, когда в них возникает необходимость. Следствием такой стратегии является возможность выполнения больших программ, размер которых может превышать размер оперативной памяти. Чтобы обеспечить данной схеме нужную производительность, отображение адресов осуществляется аппаратно при помощи многоуровневой таблицы страниц и ассоциативной памяти.

Лекция 10. Аппаратно-независимый уровень управления виртуальной памятью

Большинство ОС используют сегментно-страничную виртуальную память. Для обеспечения нужной производительности менеджер памяти ОС старается поддерживать в оперативной памяти актуальную информацию, пытаясь угадать, к каким логическим адресам последует обращение в недалеком будущем.

Ключевые слова: страничное нарушение, page fault, стратегия выборки, замещения, размещения, алгоритмы выталкивания страниц, LRU, FIFO, аномалия Belady, трешинг, модель рабочего множества.

В данной лекции рассмотрена аппаратно-независимая часть подсистемы управления виртуальной памятью, которая связана с конкретной аппаратной реализацией с помощью аппаратно-зависимой части.

Большинство ОС используют сегментно-страничную виртуальную память. Для обеспечения нужной производительности менеджер памяти ОС старается поддерживать в оперативной памяти актуальную информацию, пытаясь угадать, к каким логическим адресам последует обращение в недалеком будущем. Решающую роль здесь играет удачный выбор стратегии замещения, реализованной в алгоритме выталкивания страниц.

Исключительные ситуации при работе с памятью

Из материала предыдущей лекции следует, что отображение виртуального адреса в физический осуществляется при помощи таблицы страниц. Для каждой виртуальной страницы запись в таблице страниц содержит номер соответствующего страничного кадра в оперативной памяти, а также атрибуты страницы для контроля обращений к памяти.

Что же происходит, когда нужной страницы в памяти нет или операция обращения к памяти недопустима? Естественно, что операционная система должна быть как-то оповещена о происшедшем. Обычно для этого используется механизм исключительных ситуаций (exceptions). При попытке выполнить подобное обращение к виртуальной странице возникает исключительная ситуация *«страничное нарушение»* (page fault), приводящая к вызову специальной последовательности команд для обработки конкретного вида страничного нарушения.

Страничное нарушение может происходить в самых разных случаях: при отсутствии страницы в оперативной памяти, при попытке записи в страницу с атрибутом «только чтение» или при попытке чтения или записи

страницы с атрибутом «только выполнение». В любом из этих случаев вызывается обработчик страничного нарушения, являющийся частью операционной системы. Ему обычно передается причина возникновения исключительной ситуации и виртуальный адрес, обращение к которому вызвало нарушение.

Нас будет интересовать конкретный вариант страничного нарушения – обращение к отсутствующей странице, поскольку именно его обработка во многом определяет производительность страничной системы. Когда программа обращается к виртуальной странице, отсутствующей в основной памяти, операционная система должна выделить страницу основной памяти, переместить в нее копию виртуальной страницы из внешней памяти и модифицировать соответствующий элемент таблицы страниц.

Повышение производительности вычислительной системы может быть достигнуто за счет уменьшения частоты страничных нарушений, а также за счет увеличения скорости их обработки. Время эффективного доступа к отсутствующей в оперативной памяти странице складывается из:

- обслуживания исключительной ситуации (page fault);
- чтения (подкачки) страницы из вторичной памяти (иногда, при недостатке места в основной памяти, необходимо вытолкнуть одну из страниц из основной памяти во вторичную, то есть осуществить замещение страницы);
- возобновления выполнения процесса, вызвавшего данный page fault.

Для решения первой и третьей задач ОС выполняет до нескольких сот машинных инструкций в течение нескольких десятков микросекунд. Время подкачки страницы близко к нескольким десяткам миллисекунд. Проведенные исследования показывают, что вероятности page fault 5×10^{-7} оказывается достаточно, чтобы снизить производительность страничной схемы управления памятью на 10%. Таким образом, уменьшение частоты page faults является одной из ключевых задач системы управления памятью. Ее решение обычно связано с правильным выбором алгоритма замещения страниц.

Стратегии управления страничной памятью

Программное обеспечение подсистемы управления памятью связано с реализацией следующих стратегий:

Стратегия выборки (*fetch policy*) – в какой момент следует переписать страницу из вторичной памяти в первичную. Существует два основных варианта выборки – по запросу и с упреждением. Алгоритм выборки по запросу вступает в действие в тот момент, когда процесс обращается к отсутствующей странице, содержимое которой находится на диске. Его

реализация заключается в загрузке страницы с диска в свободную физическую страницу и коррекции соответствующей записи таблицы страниц.

Алгоритм выборки с упреждением осуществляет опережающее чтение, то есть кроме страницы, вызвавшей исключительную ситуацию, в память также загружается несколько страниц, окружающих ее (обычно соседние страницы располагаются во внешней памяти последовательно и могут быть считаны за одно обращение к диску). Такой алгоритм призван уменьшить накладные расходы, связанные с большим количеством исключительных ситуаций, возникающих при работе со значительными объемами данных или кода; кроме того, оптимизируется работа с диском.

Стратегия размещения (*placement policy*) – в какой участок первичной памяти поместить поступающую страницу. В системах со страничной организацией все просто – в любой свободный страничный кадр. В случае систем с сегментной организацией необходима стратегия, аналогичная стратегии с динамическим распределением.

Стратегия замещения (*replacement policy*) – какую страницу нужно вытолкнуть во внешнюю память, чтобы освободить место в оперативной памяти. Разумная стратегия замещения, реализованная в соответствующем алгоритме замещения страниц, позволяет хранить в памяти самую необходимую информацию и тем самым снизить частоту страничных нарушений. Замещение должно происходить с учетом выделенного каждому процессу количества кадров. Кроме того, нужно решить, должна ли замещаемая страница принадлежать процессу, который инициировал замещение, или она должна быть выбрана среди всех кадров основной памяти.

Алгоритмы замещения страниц

Итак, наиболее ответственным действием менеджера памяти является выделение кадра оперативной памяти для размещения в ней виртуальной страницы, находящейся во внешней памяти. Напомним, что мы рассматриваем ситуацию, когда размер виртуальной памяти для каждого процесса может существенно превосходить размер основной памяти. Это означает, что при выделении страницы основной памяти с большой вероятностью не удастся найти свободный страничный кадр. В этом случае операционная система в соответствии с заложенными в нее критериями должна:

- найти некоторую занятую страницу основной памяти;
- переместить в случае надобности ее содержимое во внешнюю память;
- переписать в этот страничный кадр содержимое нужной виртуальной страницы из внешней памяти;
- должным образом модифицировать необходимый элемент соответствующей таблицы страниц;

- продолжить выполнение процесса, которому эта виртуальная страница понадобилась.

Заметим, что при замещении приходится дважды передавать страницу между основной и вторичной памятью. Процесс замещения может быть оптимизирован за счет использования бита модификации (один из атрибутов страницы в таблице страниц). Бит модификации устанавливается компьютером, если хотя бы один байт был записан на страницу. При выборе кандидата на замещение проверяется бит модификации. Если бит не установлен, нет необходимости переписывать данную страницу на диск, ее копия на диске уже имеется. Подобный метод также применяется к read-only-страницам, они никогда не модифицируются. Эта схема уменьшает время обработки page fault.

Существует большое количество разнообразных алгоритмов замещения страниц. Все они делятся на локальные и глобальные. Локальные алгоритмы, в отличие от глобальных, распределяют фиксированное или динамически настраиваемое число страниц для каждого процесса. Когда процесс израсходует все предназначенные ему страницы, система будет удалять из физической памяти одну из его страниц, а не из страниц других процессов. Глобальный же алгоритм замещения в случае возникновения исключительной ситуации удовлетворится освобождением любой физической страницы, независимо от того, какому процессу она принадлежала.

Глобальные алгоритмы имеют ряд недостатков. Во-первых, они делают одни процессы чувствительными к поведению других процессов. Например, если один процесс в системе одновременно использует большое количество страниц памяти, то все остальные приложения будут в результате ощущать сильное замедление из-за недостатка кадров памяти для своей работы. Во-вторых, некорректно работающее приложение может подорвать работу всей системы (если, конечно, в системе не предусмотрено ограничение на размер памяти, выделяемой процессу), пытаясь захватить больше памяти. Поэтому в многозадачной системе иногда приходится использовать более сложные локальные алгоритмы. Применение локальных алгоритмов требует хранения в операционной системе списка физических кадров, выделенных каждому процессу. Этот список страниц иногда называют *резидентным множеством* процесса. В одном из следующих разделов рассмотрен вариант алгоритма подкачки, основанный на приведении резидентного множества в соответствие так называемому *рабочему набору* процесса.

Эффективность алгоритма обычно оценивается на конкретной последовательности ссылок к памяти, для которой подсчитывается число возникающих page faults. Эта последовательность называется *строкой обращений* (reference string). Мы можем генерировать строку обращений ис-

кусственным образом при помощи датчика случайных чисел или трассируя конкретную систему. Последний метод дает слишком много ссылок, для уменьшения числа которых можно сделать две вещи:

- для конкретного размера страниц можно запоминать только их номера, а не адреса, на которые идет ссылка;
- несколько подряд идущих ссылок на одну страницу можно фиксировать один раз.

Как уже говорилось, большинство процессоров имеют простейшие аппаратные средства, позволяющие собирать некоторую статистику обращений к памяти. Эти средства обычно включают два специальных флага на каждый элемент таблицы страниц. Флаг ссылки (reference бит) автоматически устанавливается, когда происходит любое обращение к этой странице, а уже рассмотренный выше флаг изменения (modify бит) устанавливается, если производится запись в эту страницу. Операционная система периодически проверяет установку таких флагов, для того чтобы выделить активно используемые страницы, после чего значения этих флагов сбрасываются.

Рассмотрим ряд алгоритмов замещения страниц.

Алгоритм FIFO. Выталкивание первой пришедшей страницы

Простейший алгоритм. Каждой странице присваивается временная метка. Реализуется это просто созданием очереди страниц, в конец которой страницы попадают, когда загружаются в физическую память, а из начала берутся, когда требуется освободить память. Для замещения выбирается старейшая страница. К сожалению, эта стратегия с достаточной вероятностью будет приводить к замещению активно используемых страниц, например страниц кода текстового процессора при редактировании файла. Заметим, что при замещении активных страниц все работает корректно, но page fault происходит немедленно.

Аномалия Билэди (Belady)

На первый взгляд кажется очевидным, что чем больше в памяти страничных кадров, тем реже будут иметь место page faults. Удивительно, но это не всегда так. Как установил Билэди с коллегами, определенные последовательности обращений к страницам в действительности приводят к увеличению числа страничных нарушений при увеличении кадров, выделенных процессу. Это явление носит название «аномалии Билэди» или «аномалии FIFO».

Система с тремя кадрами (9 faults) оказывается более производительной, чем с четырьмя кадрами (10 faults), для строки обращений к памяти 012301401234 при выборе стратегии FIFO.

	0	1	2	3	0	1	4	0	1	2	3	4	
Самая старая страница	0	1	2	3	0	1	4	4	4	2	3	3	
		0	1	2	3	0	1	1	1	4	2	2	
Самая новая страница			0	1	2	3	0	0	0	1	4	4	
	p	p	p	p	p	p	p			p	p		9 *page faults*

(a)

	0	1	2	3	0	1	4	0	1	2	3	4	
Самая старая страница	0	1	2	3	3	3	4	0	1	2	3	4	
		0	1	2	2	2	3	4	0	1	2	3	
			0	1	1	1	2	3	4	0	1	2	
Самая новая страница				0	0	0	1	2	3	4	0	1	
	p	p	p	p			p	p	p	p	p	p	10 *page faults*

(b)

Рис. 10.1. Аномалия Билэди: (a) – FIFO с тремя страничными кадрами; (b) – FIFO с четырьмя страничными кадрами

Аномалию Билэди следует считать скорее курьезом, чем фактором, требующим серьезного отношения, который иллюстрирует сложность ОС, где интуитивный подход не всегда приемлем.

Оптимальный алгоритм (OPT)

Одним из последствий открытия аномалии Билэди стал поиск оптимального алгоритма, который при заданной строке обращений имел бы минимальную частоту page faults среди всех других алгоритмов. Такой алгоритм был найден. Он прост: замещай страницу, которая не будет использоваться в течение самого длительного периода времени.

Каждая страница должна быть помечена числом инструкций, которые будут выполнены, прежде чем на эту страницу будет сделана первая ссылка. Выталкиваться должна страница, для которой это число наибольшее.

Этот алгоритм легко описать, но реализовать невозможно. ОС не знает, к какой странице будет следующее обращение. (Ранее такие проблемы возникали при планировании процессов – алгоритм SJF.)

Зато мы можем сделать вывод, что для того, чтобы алгоритм замещения был максимально близок к идеальному алгоритму, система должна как можно точнее предсказывать обращения процессов к памяти. Данный алгоритм применяется для оценки качества реализуемых алгоритмов.

Выталкивание дольше всего не использовавшейся страницы. Алгоритм LRU

Одним из приближений к алгоритму OPT является алгоритм, исходящий из эвристического правила, что недавнее прошлое – хороший ориентир для прогнозирования ближайшего будущего.

Ключевое отличие между FIFO и оптимальным алгоритмом заключается в том, что один смотрит назад, а другой вперед. Если использовать прошлое для аппроксимации будущего, имеет смысл замещать страницу, которая не использовалась в течение самого долгого времени. Такой подход называется least recently used алгоритм (LRU). Работа алгоритма проиллюстрирована на рис. 10.2. Сравнивая рис. 10.1 b и 10.2, можно увидеть, что использование LRU алгоритма позволяет сократить количество страничных нарушений.

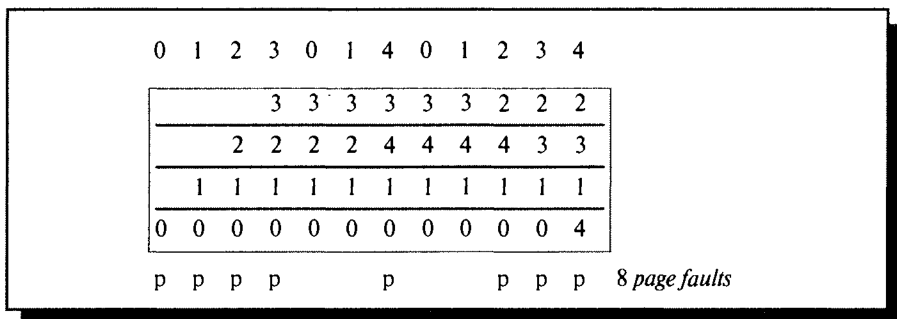


Рис. 10.2. Пример работы алгоритма LRU

LRU – хороший, но труднореализуемый алгоритм. Необходимо иметь связанный список всех страниц в памяти, в начале которого будут хранится недавно использованные страницы. Причем этот список должен обновляться при каждом обращении к памяти. Много времени нужно и на поиск страниц в таком списке.

В [Таненбаум, 2002] рассмотрен вариант реализации алгоритма LRU со специальным 64-битным указателем, который автоматически увеличивается на единицу после выполнения каждой инструкции, а в таблице страниц имеется соответствующее поле, в которое заносится значение указателя при каждой ссылке на страницу. При возникновении page fault выгружается страница с наименьшим значением этого поля.

Как оптимальный алгоритм, так и LRU не страдают от аномалии Билэди. Существует класс алгоритмов, для которых при одной и той же строке обращений множество страниц в памяти для n кадров всегда является подмножеством страниц для n+1 кадра. Эти алгоритмы не проявляют аномалии Билэди и называются стековыми (stack) алгоритмами.

Выталкивание редко используемой страницы. Алгоритм NFU

Поскольку большинство современных процессоров не предоставляют соответствующей аппаратной поддержки для реализации алгоритма LRU, хотелось бы иметь алгоритм, достаточно близкий к LRU, но не требующий специальной поддержки.

Программная реализация алгоритма, близкого к LRU, – алгоритм NFU (Not Frequently Used).

Для него требуются программные счетчики, по одному на каждую страницу, которые сначала равны нулю. При каждом прерывании по времени (а не после каждой инструкции) операционная система сканирует все страницы в памяти и у каждой страницы с установленным флагом обращения увеличивает на единицу значение счетчика, а флаг обращения сбрасывает.

Таким образом, кандидатом на освобождение оказывается страница с наименьшим значением счетчика, как страница, к которой реже всего обращались. Главный недостаток алгоритма NFU состоит в том, что он ничего не забывает. Например, страница, к которой очень часто обращались в течение некоторого времени, а потом обращаться перестали, все равно не будет удалена из памяти, потому что ее счетчик содержит большую величину. Например, в многопроходных компиляторах страницы, которые активно использовались во время первого прохода, могут надолго сохранить большие значения счетчика, мешая загрузке полезных в дальнейшем страниц.

К счастью, возможна небольшая модификация алгоритма, которая позволяет ему «забывать». Достаточно, чтобы при каждом прерывании по времени содержимое счетчика сдвигалось вправо на 1 бит, а уже затем производилось бы его увеличение для страниц с установленным флагом обращения.

Другим, уже более устойчивым недостатком алгоритма является длительность процесса сканирования таблиц страниц.

Другие алгоритмы

Для полноты картины можно упомянуть еще несколько алгоритмов.

Например, алгоритм Second-Chance – модификация алгоритма FIFO, которая позволяет избежать потери часто используемых страниц с

помощью анализа флага обращений (бита ссылки) для самой старой страницы. Если флаг установлен, то страница, в отличие от алгоритма FIFO, не выталкивается, а ее флаг сбрасывается, и страница переносится в конец очереди. Если первоначально флаги обращений были установлены для всех страниц (на все страницы ссылались), алгоритм Second-Chance превращается в алгоритм FIFO. Данный алгоритм использовался в Multics и BSD Unix.

В компьютере Macintosh использован алгоритм NRU (Not Recently-Used), где страница-«жертва» выбирается на основе анализа битов модификации и ссылки. Интересные стратегии, основанные на буферизации страниц, реализованы в VAX/VMS и Mach.

Имеется также и много других алгоритмов замещения. Объем этого курса не позволяет рассмотреть их подробно. Подробное описание различных алгоритмов замещения можно найти в монографиях [Дейтел, 1987], [Цикритис, 1977], [Таненбаум, 2002] и др.

Управление количеством страниц, выделенным процессу. Модель рабочего множества

В стратегиях замещения, рассмотренных в предыдущем разделе, прослеживается предположение о том, что количество кадров, принадлежащих процессу, нельзя увеличить. Это приводит к необходимости выталкивания страницы. Рассмотрим более общий подход, базирующийся на концепции рабочего множества, сформулированной Деннингом [Denning, 1996].

Итак, что делать, если в распоряжении процесса имеется недостаточное число кадров? Нужно ли его приостановить с освобождением всех кадров? Что следует понимать под достаточным количеством кадров?

Трешинг (Thrashing)

Хотя теоретически возможно уменьшить число кадров процесса до минимума, существует какое-то число активно используемых страниц, без которого процесс часто генерирует page faults. Высокая частота страничных нарушений называется трешинг (thrashing, иногда употребляется русский термин «пробуксовка», см. рис. 10.3). Процесс находится в состоянии трешинга, если при его работе больше времени уходит на подкачку страниц, нежели на выполнение команд. Такого рода критическая ситуация возникает вне зависимости от конкретных алгоритмов замещения.

Часто результатом трешинга является снижение производительности вычислительной системы. Один из нежелательных сценариев развития событий может выглядеть следующим образом. При глобальном алгорит-

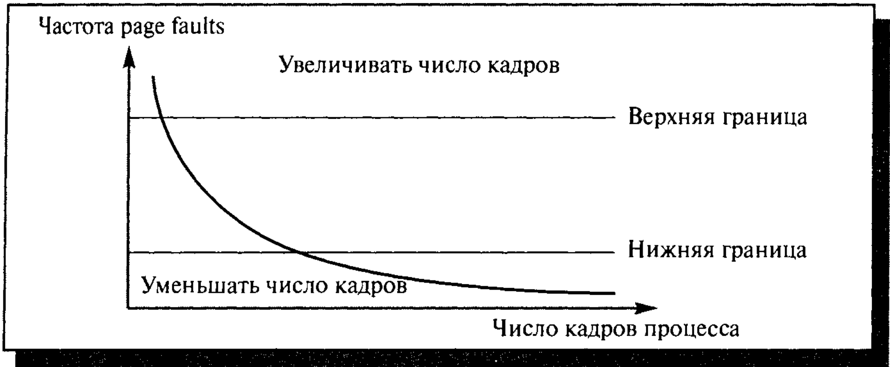


Рис. 10.3. Частота page faults в зависимости от количества кадров, выделенных процессу

ме замещения процесс, которому не хватает кадров, начинает отбирать кадры у других процессов, которые в свою очередь начинают заниматься тем же. В результате все процессы попадают в очередь запросов к устройству вторичной памяти (находятся в состоянии ожидания), а очередь процессов в состоянии готовности пустеет. Загрузка процессора снижается. Операционная система реагирует на это увеличением степени мультипрограммирования, что приводит к еще большему трешингу и дальнейшему снижению загрузки процессора. Таким образом, пропускная способность системы падает из-за трешинга.

Эффект трешинга, возникающий при использовании глобальных алгоритмов, может быть ограничен за счет применения локальных алгоритмов замещения. При локальных алгоритмах замещения если даже один из процессов попал в трешинг, это не сказывается на других процессах. Однако он много времени проводит в очереди к устройству выгрузки, затрудняя подкачку страниц остальных процессов.

Критическая ситуация типа трешинга возникает вне зависимости от конкретных алгоритмов замещения. Единственным алгоритмом, теоретически гарантирующим отсутствие трешинга, является рассмотренный выше не реализуемый на практике оптимальный алгоритм.

Итак, трешинг – это высокая частота страничных нарушений. Необходимо ее контролировать. Когда она высока, процесс нуждается в кадрах. Можно, устанавливая желаемую частоту page faults, регулировать размер процесса, добавляя или отнимая у него кадры. Может оказаться целесообразным выгрузить процесс целиком. Освободившиеся кадры выделяются другим процессам с высокой частотой page faults.

Для предотвращения трешинга требуется выделять процессу столько кадров, сколько ему нужно. Но как узнать, сколько ему нужно? Необходимо попытаться выяснить, как много кадров процесс ре-

ально использует. Для решения этой задачи Деннинг использовал модель рабочего множества, которая основана на применении принципа локальности.

Модель рабочего множества

Рассмотрим поведение реальных процессов.

Процессы начинают работать, не имея в памяти необходимых страниц. В результате при выполнении первой же машинной инструкции возникает page fault, требующий подкачки порции кода. Следующий page fault происходит при локализации глобальных переменных и еще один – при выделении памяти для стека. После того как процесс собрал большую часть необходимых ему страниц, page faults возникают редко.

Таким образом, существует набор страниц (P_1, P_2, ... P_n), активно использующихся вместе, который позволяет процессу в момент времени t в течение некоторого периода T производительно работать, избегая большого количества page faults. Этот набор страниц называется *рабочим множеством* W(t,T) (*working set*) процесса. Число страниц в рабочем множестве определяется параметром T, является неубывающей функцией T и относительно невелико. Иногда T называют размером окна рабочего множества, через которое ведется наблюдение за процессом (см. рис. 10.4).

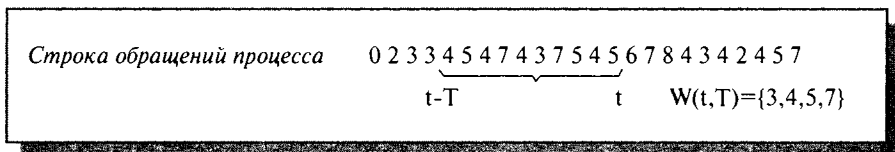


Рис. 10.4. Пример рабочего множества процесса

Легко написать тестовую программу, которая систематически работает с большим диапазоном адресов, но, к счастью, большинство реальных процессов не ведут себя подобным образом, а проявляют свойство локальности. В течение любой фазы вычислений процесс работает с небольшим количеством страниц.

Когда процесс выполняется, он двигается от одного рабочего множества к другому. Программа обычно состоит из нескольких рабочих множеств, которые могут перекрываться. Например, когда вызвана процедура, она определяет новое рабочее множество, состоящее из страниц, содержащих инструкции процедуры, ее локальные и глобальные переменные. После ее завершения процесс покидает это рабочее множество, но может вернуться к нему при новом вызове процедуры. Таким образом, рабочее множество определяется кодом и данными

программы. Если процессу выделять меньше кадров, чем ему требуется для поддержки рабочего множества, он будет находиться в состоянии трешинга.

Принцип локальности ссылок препятствует частым изменениям рабочих наборов процессов. Формально это можно выразить следующим образом. Если в период времени (t – T, t) программа обращалась к страницам W(t,T), то при надлежащем выборе T с большой вероятностью эта программа будет обращаться к тем же страницам в период времени (t, t + T). Другими словами, принцип локальности утверждает, что если не слишком далеко заглядывать в будущее, то можно достаточно точно его прогнозировать исходя из прошлого. Понятно, что с течением времени рабочий набор процесса может изменяться (как по составу страниц, так и по их числу).

Наиболее важное свойство рабочего множества – его размер. ОС должна выделить каждому процессу достаточное число кадров, чтобы поместилось его рабочее множество. Если кадры еще остались, то может быть инициирован другой процесс. Если рабочие множества процессов не помещаются в память и начинается трешинг, то один из процессов можно выгрузить на диск.

Решение о размещении процессов в памяти должно, следовательно, базироваться на размере его рабочего множества. Для впервые инициируемых процессов это решение может быть принято эвристически. Во время работы процесса система должна уметь определять: расширяет процесс свое рабочее множество или перемещается на новое рабочее множество. Если в состав атрибутов страницы включить время последнего использования t_i (для страницы с номером i), то принадлежность i-й страницы к рабочему набору, определяемому параметром t в момент времени t будет выражаться неравенством: $t - T < t_i < t$. Алгоритм выталкивания страниц WSClock, использующий информацию о рабочем наборе процесса, описан в [Таненбаум, 2002].

Другой способ реализации данного подхода может быть основан на отслеживании количества страничных нарушений, вызываемых процессом. Если процесс часто генерирует page faults и память не слишком заполнена, то система может увеличить число выделенных ему кадров. Если же процесс не вызывает исключительных ситуаций в течение некоторого времени и уровень генерации ниже какого-то порога, то число кадров процесса может быть урезано. Этот способ регулирует лишь размер множества страниц, принадлежащих процессу, и должен быть дополнен какой-либо стратегией замещения страниц. Несмотря на то что система при этом может пробуксовывать в моменты перехода от одного рабочего множества к другому, предлагаемое решение в состоянии обеспечить наилучшую производительность для каждого процесса, не требуя никакой дополнительной настройки системы.

Страничные демоны

Подсистема виртуальной памяти работает производительно при наличии резерва свободных страничных кадров. Алгоритмы, обеспечивающие поддержку системы в состоянии отсутствия трешинга, реализованы в составе фоновых процессов (их часто называют демонами или сервисами), которые периодически «просыпаются» и инспектируют состояние памяти. Если свободных кадров оказывается мало, они могут сменить стратегию замещения. Их задача – поддерживать систему в состоянии наилучшей производительности.

Примером такого рода процесса может быть фоновый процесс – сборщик страниц, реализующий облегченный вариант алгоритма откачки, основанный на использовании рабочего набора и применяемый во многих клонах ОС Unix (см., например, [Bach, 1986]). Данный демон производит откачку страниц, не входящих в рабочие наборы процессов. Он начинает активно работать, когда количество страниц в списке свободных страниц достигает установленного нижнего порога, и пытается выталкивать страницы в соответствии с собственной стратегией.

Но если возникает требование страницы в условиях, когда список свободных страниц пуст, то начинает работать механизм свопинга, поскольку простое отнятие страницы у любого процесса (включая тот, который затребовал бы страницу) потенциально вело бы к ситуации thrashing, и разрушало бы рабочий набор некоторого процесса. Любой процесс, затребовавший страницу не из своего текущего рабочего набора, становится в очередь на выгрузку в расчете на то, что после завершения выгрузки хотя бы одного из процессов свободной памяти уже может быть достаточно.

В ОС Windows 2000 аналогичную роль играет менеджер балансного набора (Working set manager), который вызывается раз в секунду или тогда, когда размер свободной памяти опускается ниже определенного предела, и отвечает за суммарную политику управления памятью и поддержку рабочих множеств.

Программная поддержка сегментной модели памяти процесса

Реализация функций операционной системы, связанных с поддержкой памяти, – ведение таблиц страниц, трансляция адреса, обработка страничных ошибок, управление ассоциативной памятью и др. – тесно связана со структурами данных, обеспечивающими удобное представление адресного пространства процесса. Формат этих структур сильно зависит от аппаратуры и особенностей конкретной ОС.

Чаще всего виртуальная память процесса ОС разбивается на сегмен ты пяти типов: кода программы, данных, стека, разделяемый и сегмен файлов, отображаемых в память (см. рис. 10.5).

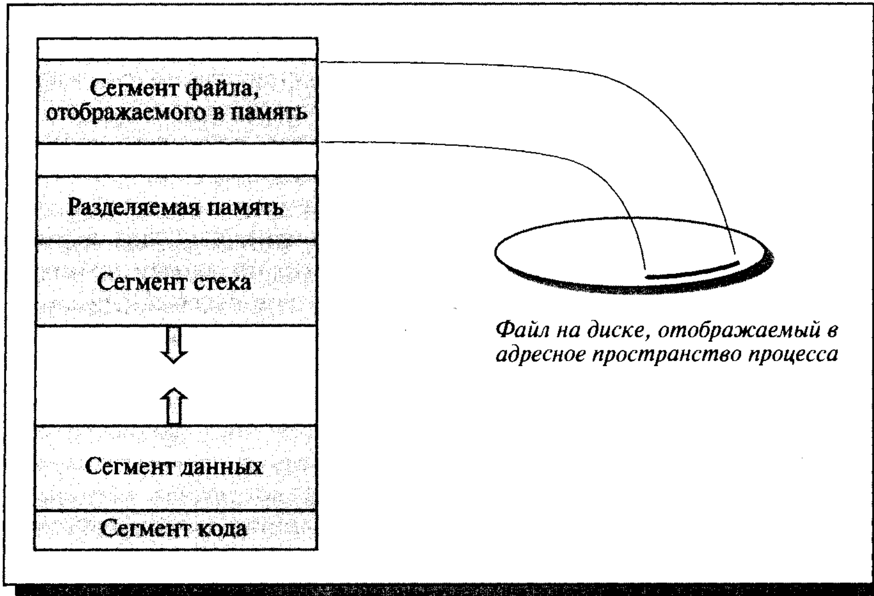


Рис. 10.5. Образ процесса в памяти

Сегмент программного кода содержит только команды. Сегмент про граммного кода не модифицируется в ходе выполнения процесса, обычн страницы данного сегмента имеют атрибут read-only. Следствием этог является возможность использования одного экземпляра кода для разны процессов.

Сегмент данных, содержащий переменные программы и сегмент стека содержащий автоматические переменные, могут динамически менять сво размер (обычно данные в сторону увеличения адресов, а стек – в сторон уменьшения) и содержимое, должны быть доступны по чтению и запис и являются приватными сегментами процесса.

С целью обобществления памяти между несколькими процессам создаются *разделяемые сегменты*, допускающие доступ по чтению и запи си. Вариантом разделяемого сегмента может быть *сегмент файла, отобра жаемого в память*. Специфика таких сегментов состоит в том, что из ни откачка осуществляется не в системную область выгрузки, а непосредст венно в отображаемый файл. Реализация разделяемых сегментов основан на том, что логические страницы различных процессов связываются с од ними и теми же страничными кадрами.

Сегменты представляют собой непрерывные области (в Linux они так и называются – области) в виртуальном адресном пространстве процесса, выровненные по границам страниц. Каждая область состоит из набора страниц с одним и тем же режимом защиты. Между областями в виртуальном пространстве могут быть свободные участки. Естественно, что подобные объекты описаны соответствующими структурами (см., например, структуры `mm_struct` и `vm_area_struct` в Linux).

Часть работы по организации сегментов может происходить с участием программиста. Особенно это заметно при низкоуровневом программировании. В частности, отдельные области памяти могут быть поименованы и использоваться для обмена данными между процессами. Два процесса могут общаться через разделяемую область памяти при условии, что им известно ее имя (пароль). Обычно это делается при помощи специальных вызовов (например, map и unmap), входящих в состав интерфейса виртуальной памяти.

Загрузка исполняемого файла (системный вызов exec) осуществляется обычно через отображение (mapping) его частей (кода, данных) в соответствующие сегменты адресного пространства процесса. Например, сегмент кода является сегментом отображаемого в память файла, содержащего исполняемую программу. При попытке выполнить первую же инструкцию система обнаруживает, что нужной части кода в памяти нет, генерирует page fault и подкачивает эту часть кода с диска. Далее процедура повторяется до тех пор, пока вся программа не окажется в оперативной памяти.

Как уже говорилось, размер сегмента данных динамически меняется. Рассмотрим, как организована поддержка сегментов данных в Unix. Пользователь, запрашивая (библиотечные вызовы malloc, new) или освобождая (free, delete) память для динамических данных, фактически изменяет границу выделенной процессу памяти через системный вызов brk (от слова break), который модифицирует значение переменной brk из структуры данных процесса. В результате происходит выделение физической памяти, граница brk смещается в сторону увеличения виртуальных адресов, а соответствующие строки таблиц страниц получают осмысленные значения. При помощи того же вызова brk пользователь может уменьшить размер сегмента данных. На практике освобожденная пользователем виртуальная память (библиотечные вызовы free, delete) системе не возвращается. На это есть две причины. Во-первых, для уменьшения размеров сегмента данных необходимо организовать его уплотнение или «сборку мусора». А во-вторых, незанятые внутри сегмента данных области естественным образом будут вытолкнуты из оперативной памяти вследствие того, что к ним не будет обращений. Ведение списков занятых и свободных областей памяти в сегменте данных пользователя осуществляется на уровне системных библиотек.

Более подробно информация об адресных пространствах процессов в Unix изложена в [Кузнецов], [Bach, 1986].

Отдельные аспекты функционирования менеджера памяти

Корректная работа менеджера памяти помимо принципиальных вопросов, связанных с выбором абстрактной модели виртуальной памяти и ее аппаратной поддержкой, обеспечивается также множеством нюансов и мелких деталей. В качестве примера такого рода компонента рассмотрим более подробно *локализацию страниц в памяти*, которая применяется в тех случаях, когда поддержка страничной системы приводит к необходимости разрешить определенным страницам, хранящим буферы ввода-вывода, другие важные данные и код, быть блокированными в памяти.

Рассмотрим случай, когда система виртуальной памяти может вступить в конфликт с подсистемой ввода-вывода. Например, процесс может запросить ввод в буфер и ожидать его завершения. Управление передастся другому процессу, который может вызвать page fault и, с отличной от нуля вероятностью, спровоцировать выгрузку той страницы, куда должен быть осуществлен ввод первым процессом. Подобные ситуации нуждаются в дополнительном контроле, особенно если ввод-вывод реализован с использованием механизма прямого доступа к памяти (DMA). Одно из решений данной проблемы – вводить данные в невытесняемый буфер в пространстве ядра, а затем копировать их в пользовательское пространство.

Второе решение – локализовать страницы в памяти, используя специальный бит локализации, входящий в состав атрибутов страницы. Локализованная страница замещению не подлежит. Бит локализации сбрасывается после завершения операции ввода-вывода.

Другое использование бита локализации может иметь место и при нормальном замещении страниц. Рассмотрим следующую цепь событий. Низкоприоритетный процесс после длительного ожидания получил в свое распоряжение процессор и подкачал с диска нужную ему страницу. Если он сразу после этого будет вытеснен высокоприоритетным процессом, последний может легко заместить вновь подкачанную страницу низкоприоритетного, так как на нее не было ссылок. Имеет смысл вновь загруженные страницы помечать битом локализации до первой ссылки, иначе низкоприоритетный процесс так и не начнет работать.

Использование бита локализации может быть опасным, если забыть его отключить. Если такая ситуация имеет место, страница становится неиспользуемой. SunOS разрешает использование данного бита в качестве подсказки, которую можно игнорировать, когда пул свободных кадров становится слишком маленьким.

Другим важным применением локализации является ее использование в системах мягкого *реального времени*. Рассмотрим процесс или нить реального времени. Вообще говоря, виртуальная память – антитеза вычислений реального времени, так как дает непредсказуемые задержки при подкачке страниц. Поэтому системы реального времени почти не используют виртуальную память. ОС Solaris поддерживает как реальное время, так и разделение времени. Для решения проблемы page faults Solaris разрешает процессам сообщать системе, какие страницы важны для процесса, и локализовать их в памяти. В результате возможно выполнение процесса, реализующего задачу реального времени, содержащего локализованные страницы, где временные задержки страничной системы будут минимизированы.

Помимо системы локализации страниц, есть и другие интересные проблемы, возникающие в процессе управления памятью. Так, например, бывает непросто осуществить повторное выполнение инструкции, вызвавшей page fault. Представляют интерес и алгоритмы отложенного выделения памяти (копирование при записи и др.). Ограниченный объем данного курса не позволяет рассмотреть их более подробно.

Заключение

Описанная система управления памятью является совокупностью программно-технических средств, обеспечивающих производительное функционирование современных компьютеров. Успех реализации той части ОС, которая относится к управлению виртуальной памятью, определяется близостью архитектуры аппаратных средств, поддерживающих виртуальную память, к абстрактной модели виртуальной памяти ОС. Справедливости ради заметим, что в подавляющем большинстве современных компьютеров аппаратура выполняет функции, существенно превышающие потребности модели ОС, так что создание аппаратно-зависимой части подсистемы управления виртуальной памятью ОС в большинстве случаев не является чрезмерно сложной задачей.

Часть IV. Файловые системы

Все компьютерные приложения нуждаются в хранении и обновлении информации. Возможности оперативной памяти для хранения информации ограничены. Во-первых, оперативная память обычно теряет свое содержимое после отключения питания, а во-вторых, объем обрабатываемых данных зачастую превышает ее возможности. Кроме того, информацию желательно иметь в виде, не зависящем от процессов. Поэтому принято хранить данные на внешних носителях (обычно это диски) в единицах, называемых файлами. В большинстве компьютерных систем предусмотрены устройства внешней (вторичной) памяти большой емкости, на которых можно хранить огромные объемы данных. Однако характеристики доступа к таким устройствам существенно отличаются от характеристик доступа к основной памяти. Чтобы повысить эффективность использования этих устройств, был разработан ряд специфичных для них структур данных и алгоритмов.

Лекция 11. Файлы с точки зрения пользователя

В настоящей лекции вводится понятие и рассматриваются основные функции и интерфейс файловой системы.

Ключевые слова: файл, внешняя память, индексация, тип файла, атрибуты файла, доступ к файлу, прямой, последовательный, индексно-последовательный файл, каталоги, директории, дерево каталогов, операции над файлами, операции над каталогами, защита файлов.

Введение

История систем управления данными во внешней памяти начинается еще с магнитных лент, но современный облик они приобрели с появлением магнитных дисков. До этого каждая прикладная программа сама решала проблемы именования данных и их структуризации во внешней памяти. Это затрудняло поддержание на внешнем носителе нескольких архивов долговременно хранящейся информации. Историческим шагом стал переход к использованию централизованных систем управления файлами. Система управления файлами берет на себя распределение внешней памяти, отображение имен файлов в адреса внешней памяти и обеспечение доступа к данным.

Файловая система – это часть операционной системы, назначение которой состоит в том, чтобы организовать эффективную работу с данными, хранящимися во внешней памяти, и обеспечить пользователю удобный интерфейс при работе с такими данными. Организовать хранение информации на магнитном диске непросто. Это требует, например, хорошего знания устройства контроллера диска, особенностей работы с его регистрами. Непосредственное взаимодействие с диском – прерогатива компонента системы ввода-вывода ОС, называемого драйвером диска. Для того чтобы избавить пользователя компьютера от сложностей взаимодействия с аппаратурой, была придумана ясная абстрактная модель файловой системы. Операции записи или чтения файла концептуально проще, чем низкоуровневые операции работы с устройствами.

Основная идея использования внешней памяти состоит в следующем. ОС делит память на блоки фиксированного размера, например, 4096 байт. Файл, обычно представляющий собой неструктурированную последовательность однобайтовых записей, хранится в виде последовательности блоков (не обязательно смежных); каждый блок хранит целое число записей. В некоторых ОС (MS-DOS) адреса блоков, содержащих данные файла, могут быть организованы в связный список и вынесены в отдельную таблицу в памяти. В других ОС (Unix) адреса блоков данных файла хранятся в отдельном блоке внешней памяти (так называемом индексе или индексном узле). Этот прием, называемый *индексацией*, является наиболее распространенным для приложений, требующих произвольного доступа к записям файлов. Индекс файла состоит из списка элементов, каждый из которых содержит номер блока в файле и сведения о местоположении данного блока. Считывание очередного байта осуществляется с так называемой *текущей* позиции, которая характеризуется смещением от начала файла. Зная размер блока, легко вычислить номер блока, содержащего текущую позицию. Адрес же нужного блока диска можно затем извлечь из индекса файла. Базовой операцией, выполняемой по отношению к файлу, является чтение блока с диска и перенос его в буфер, находящийся в основной памяти.

Файловая система позволяет при помощи системы справочников (каталогов, директорий) связать уникальное имя файла с блоками вторичной памяти, содержащими данные файла. Иерархическая структура каталогов, используемая для управления файлами, может служить другим примером индексной структуры. В этом случае каталоги или папки играют роль индексов, каждый из которых содержит ссылки на свои подкаталоги. С этой точки зрения вся файловая система компьютера представляет собой большой индексированный файл. Помимо собственно файлов и структур данных, используемых для управления файлами (каталоги,

дескрипторы файлов, различные таблицы распределения внешней памяти), понятие «файловая система» включает программные средства, реализующие различные операции над файлами.

Перечислим *основные функции* файловой системы:

1) Идентификация файлов. Связывание имени файла с выделенным ему пространством внешней памяти.
2) Распределение внешней памяти между файлами. Для работы с конкретным файлом пользователю не требуется иметь информацию о местоположении этого файла на внешнем носителе информации. Например, для того чтобы загрузить документ в редактор с жесткого диска, нам не нужно знать, на какой стороне какого магнитного диска, на каком цилиндре и в каком секторе находится данный документ.
3) Обеспечение надежности и отказоустойчивости. Стоимость информации может во много раз превышать стоимость компьютера.
4) Обеспечение защиты от несанкционированного доступа.
5) Обеспечение совместного доступа к файлам, так чтобы пользователю не приходилось прилагать специальных усилий по обеспечению синхронизации доступа.
6) Обеспечение высокой производительности.

Иногда говорят, что файл – это поименованный набор связанной информации, записанной во вторичную память. Для большинства пользователей файловая система – наиболее видимая часть ОС. Она предоставляет механизм для онлайнового хранения и доступа как к данным, так и к программам для всех пользователей системы. С точки зрения пользователя, файл – единица внешней памяти, то есть данные, записанные на диск, должны быть в составе какого-нибудь файла.

Важный аспект организации файловой системы – учет стоимости операций взаимодействия с вторичной памятью. Процесс считывания блока диска состоит из позиционирования считывающей головки над дорожкой, содержащей требуемый блок, ожидания, пока требуемый блок сделает оборот и окажется под головкой, и собственно считывания блока. Для этого требуется значительное время (десятки миллисекунд). В современных компьютерах обращение к диску осуществляется примерно в 100 000 раз медленнее, чем обращение к оперативной памяти. Таким образом, критерием вычислительной сложности алгоритмов, работающих с внешней памятью, является количество обращений к диску.

В данной лекции рассматриваются вопросы структуры, именования, защиты файлов; операции, которые разрешается производить над файлами; организация файлового архива (полного дерева справочников). Проблемы выделения дискового пространства, обеспечения производительной работы файловой системы и ряд других вопросов, интересующих разработчиков системы, вы найдете в следующей лекции.

Общие сведения о файлах

Имена файлов

Файлы представляют собой абстрактные объекты. Их задача – хранить информацию, скрывая от пользователя детали работы с устройствами. Когда процесс создает файл, он дает ему имя. После завершения процесса файл продолжает существовать и через свое имя может быть доступен другим процессам.

Правила именования файлов зависят от ОС. Многие ОС поддерживают имена из *двух частей* (имя + расширение), например `progr.c` (файл, содержащий текст программы на языке Си) или `autoexec.bat` (файл, содержащий команды интерпретатора командного языка). Тип расширения файла позволяет ОС организовать работу с ним различных прикладных программ в соответствии с заранее оговоренными соглашениями. Обычно ОС накладывают некоторые ограничения, как на используемые в имени символы, так и на длину имени файла. В соответствии со стандартом POSIX, популярные ОС оперируют удобными для пользователя длинными именами (до 255 символов).

Типы файлов

Важный аспект организации файловой системы и ОС – следует ли поддерживать и распознавать типы файлов. Если да, то это может помочь правильному функционированию ОС, например не допустить вывода на принтер бинарного файла.

Основные типы файлов: регулярные (обычные) файлы и директории (справочники, каталоги). Обычные файлы содержат пользовательскую информацию. Директории – системные файлы, поддерживающие структуру файловой системы. В каталоге содержится перечень входящих в него файлов и устанавливается соответствие между файлами и их характеристиками (атрибутами). Мы будем рассматривать директории ниже.

Напомним, что хотя внутри подсистемы управления файлами обычный файл представляется в виде набора блоков внешней памяти, для пользователей обеспечивается представление файла в виде линейной последовательности байтов. Такое представление позволяет использовать абстракцию файла при работе с внешними устройствами, при организации межпроцессных взаимодействий и т. д. Так, например, клавиатура обычно рассматривается как текстовый файл, из которого компьютер получает данные в символьном формате. Поэтому иногда к файлам приписывают другие объекты ОС, например специальные символьные файлы и специальные блочные файлы, именованные каналы и сокеты, имеющие

файловый интерфейс. Эти объекты рассматриваются в других разделах данного курса.

Далее речь пойдет главным образом об *обычных файлах.*

Обычные (или регулярные) файлы реально представляют собой набор блоков (возможно, пустой) на устройстве внешней памяти, на котором поддерживается файловая система. Такие файлы могут содержать как текстовую информацию (обычно в формате ASCII), так и произвольную двоичную (бинарную) информацию.

Текстовые файлы содержат символьные строки, которые можно распечатать, увидеть на экране или редактировать обычным текстовым редактором.

Другой тип файлов – нетекстовые, или бинарные, файлы. Обычно они имеют некоторую внутреннюю структуру. Например, исполняемый файл в ОС Unix имеет пять секций: заголовок, текст, данные, биты реаллокации и символьную таблицу. ОС выполняет файл, только если он имеет нужный формат. Другим примером бинарного файла может быть архивный файл. Типизация файлов не слишком строгая.

Обычно прикладные программы, работающие с файлами, распознают тип файла по его имени в соответствии с общепринятыми соглашениями. Например, файлы с расширениями `.c`, `.pas`, `.txt` – ASCII-файлы, файлы с расширениями `.exe` – выполнимые, файлы с расширениями `.obj`, `.zip` – бинарные и т. д.

Атрибуты файлов

Кроме имени ОС часто связывают с каждым файлом и другую информацию, например дату модификации, размер и т. д. Эти другие характеристики файлов называются ***атрибутами.*** Список атрибутов в разных ОС может варьироваться. Обычно он содержит следующие элементы: основную информацию (имя, тип файла), адресную информацию (устройство, начальный адрес, размер), информацию об управлении доступом (владелец, допустимые операции) и информацию об использовании (даты создания, последнего чтения, модификации и др.).

Список атрибутов обычно хранится в структуре директорий (см. следующую лекцию) или других структурах, обеспечивающих доступ к данным файла.

Организация файлов и доступ к ним

Программист воспринимает файл в виде набора однородных записей. Запись – это наименьший элемент данных, который может быть обработан как единое целое прикладной программой при обмене с внеш-

ним устройством. Причем в большинстве ОС размер записи равен одному байту. В то время как приложения оперируют записями, физический обмен с устройством осуществляется большими единицами (обычно блоками). Поэтому записи объединяются в блоки для вывода и разблокируются – для ввода. Вопросы распределения блоков внешней памяти между файлами рассматриваются в следующей лекции.

ОС поддерживают несколько вариантов структуризации файлов.

Последовательный файл

Простейший вариант – так называемый последовательный файл. То есть файл является последовательностью записей. Поскольку записи, как правило, однобайтовые, файл представляет собой *неструктурированную последовательность байтов*.

Обработка подобных файлов предполагает последовательное чтение записей от начала файла, причем конкретная запись определяется ее положением в файле. Такой способ доступа называется *последовательным* (модель ленты). Если в качестве носителя файла используется магнитная лента, то так и делается. Текущая позиция считывания может быть возвращена к началу файла (rewind).

Файл прямого доступа

В реальной практике файлы хранятся на устройствах *прямого* (random) доступа, например на дисках, поэтому содержимое файла может быть разбросано по разным блокам диска, которые можно считывать в произвольном порядке. Причем номер блока однозначно определяется позицией внутри файла.

Здесь имеется в виду относительный номер, специфицирующий данный блок среди блоков диска, принадлежащих файлу. О связи относительного номера блока с абсолютным его номером на диске рассказывается в следующей лекции.

Естественно, что в этом случае для доступа к середине файла просмотр всего файла с самого начала не обязателен. Для специфицирования места, с которого надо начинать чтение, используются два способа: с начала или с текущей позиции, которую дает операция seek. Файл, байты которого могут быть считаны в произвольном порядке, называется *файлом прямого доступа*.

Таким образом, файл, состоящий из однобайтовых записей на устройстве прямого доступа, – наиболее распространенный способ организации файла. Базовыми операциями для такого рода файлов являются считывание или запись символа в текущую позицию. В большинстве язы-

ков высокого уровня предусмотрены операторы посимвольной пересылки данных в файл или из него.

Подобную логическую структуру имеют файлы во многих файловых системах, например в файловых системах ОС Unix и MS-DOS. ОС не осуществляет никакой интерпретации содержимого файла. Эта схема обеспечивает максимальную гибкость и универсальность. С помощью базовых системных вызовов (или функций библиотеки ввода/вывода) пользователи могут как угодно структурировать файлы. В частности, многие СУБД хранят свои базы данных в обычных файлах.

Другие формы организации файлов

Известны как другие формы организации файла, так и другие способы доступа к ним, которые использовались в ранних ОС, а также применяются сегодня в больших мэйнфреймах (mainframe), ориентированных на коммерческую обработку данных.

Первый шаг в структурировании – хранение файла в виде *последовательности записей фиксированной длины*, каждая из которых имеет внутреннюю структуру. Операция чтения производится над записью, а операция записи переписывает или добавляет запись целиком. Ранее использовались записи по 80 байт (это соответствовало числу позиций в перфокарте) или по 132 символа (ширина принтера). В ОС CP/M файлы были последовательностями 128-символьных записей. С введением CRT-терминалов данная идея утратила популярность.

Другой способ представления файлов – *последовательность записей переменной длины*, каждая из которых содержит ключевое поле в фиксированной позиции внутри записи (см. рис. 11.1). Базисная операция в данном случае – считать запись с каким-либо значением ключа. Записи могут располагаться в файле последовательно (например, отсортированные по значению ключевого поля) или в более сложном порядке. Метод доступа по значению ключевого поля к записям последовательного файла называется *индексно-последовательным*.

В некоторых системах ускорение доступа к файлу обеспечивается конструированием *индекса* файла. Индекс обычно хранится на том же

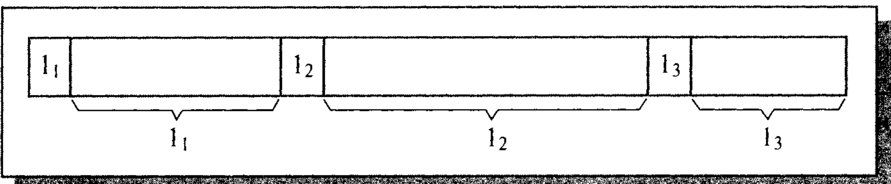


Рис. 11.1. Файл как последовательность записей переменной длины

устройстве, что и сам файл, и состоит из списка элементов, каждый из которых содержит идентификатор записи, за которым следует указание о местоположении данной записи. Для поиска записи вначале происходит обращение к индексу, где находится указатель на нужную запись. Такие файлы называются *индексированными*, а метод доступа к ним – *доступ с использованием индекса.*

Предположим, у нас имеется большой несортированный файл, содержащий разнообразные сведения о студентах, состоящие из записей с несколькими полями, и возникает задача организации быстрого поиска по одному из полей, например по фамилии студента. Рис. 11.2 иллюстрирует решение данной проблемы – организацию метода доступа к файлу с использованием индекса.

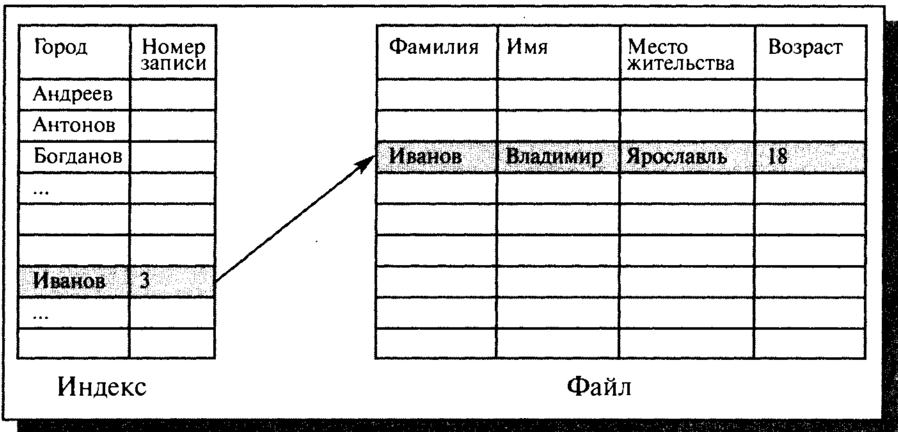


Рис. 11.2. Пример организации индекса для последовательного файла

Следует отметить, что почти всегда главным фактором увеличения скорости доступа является *избыточность* данных.

Способ выделения дискового пространства при помощи индексных узлов, применяемый в ряде ОС (Unix и некоторых других, см. следующую лекцию), может служить другим примером организации индекса.

В этом случае ОС использует древовидную организацию блоков, при которой блоки, составляющие файл, являются листьями дерева, а каждый внутренний узел содержит указатели на множество блоков файла. Для больших файлов индекс может быть слишком велик. В этом случае создают индекс для индексного файла (блоки промежуточного уровня или блоки косвенной адресации).

Операции над файлами

Операционная система должна предоставить в распоряжение пользователя набор операций для работы с файлами, реализованных через системные вызовы. Чаще всего при работе с файлом пользователь выполняет не одну, а несколько операций. Во-первых, нужно найти данные файла и его атрибуты по символьному имени, во-вторых, считать необходимые атрибуты файла в отведенную область оперативной памяти и проанализировать права пользователя на выполнение требуемой операции. Затем следует выполнить операцию, после чего освободить занимаемую данными файла область памяти. Рассмотрим в качестве примера основные файловые операции ОС Unix [Таненбаум, 2002]:

- Создание файла, не содержащего данных. Смысл данного вызова – объявить, что файл существует, и присвоить ему ряд атрибутов. При этом выделяется место для файла на диске и вносится запись в каталог.
- Удаление файла и освобождение занимаемого им дискового пространства.
- Открытие файла. Перед использованием файла процесс должен его открыть. Цель данного системного вызова – разрешить системе проанализировать атрибуты файла и проверить права доступа к нему, а также считать в оперативную память список адресов блоков файла для быстрого доступа к его данным. Открытие файла является процедурой создания *дескриптора* или управляющего блока файла. Дескриптор (описатель) файла хранит всю информацию о нем. Иногда, в соответствии с парадигмой, принятой в языках программирования, под дескриптором понимается альтернативное имя файла или указатель на описание файла в таблице открытых файлов, используемый при последующей работе с файлом. Например, на языке Си операция открытия файла `fd=open(pathname,flags,modes);` возвращает дескриптор `fd`, который может быть задействован при выполнении операций чтения (`read(fd,buffer,count);`) или записи.
- Закрытие файла. Если работа с файлом завершена, его атрибуты и адреса блоков на диске больше не нужны. В этом случае файл нужно закрыть, чтобы освободить место во внутренних таблицах файловой системы.
- Позиционирование. Дает возможность специфицировать место внутри файла, откуда будет производиться считывание (или запись) данных, то есть задать *текущую* позицию.
- Чтение данных из файла. Обычно это делается с текущей позиции. Пользователь должен задать объем считываемых данных и предоставить для них буфер в оперативной памяти.

- Запись данных в файл с текущей позиции. Если текущая позиция находится в конце файла, его размер увеличивается, в противном случае запись осуществляется на место имеющихся данных, которые, таким образом, теряются.

Есть и другие операции, например переименование файла, получение атрибутов файла и т. д.

Существует два способа выполнить последовательность действий над файлами [Олифер, 2001].

В первом случае для каждой операции выполняются как универсальные, так и уникальные действия (схема stateless). Например, последовательность операций может быть такой: open, read1, close, ... open, read2, close, ... open, read3, close.

Альтернативный способ – это когда универсальные действия выполняются в начале и в конце последовательности операций, а для каждой промежуточной операции выполняются только уникальные действия. В этом случае последовательность вышеприведенных операций будет выглядеть так: open, read1, ... read2, ... read3, close.

Большинство ОС использует второй способ, более экономичный и быстрый. Первый способ более устойчив к сбоям, поскольку результаты каждой операции становятся независимыми от результатов предыдущей операции; поэтому он иногда применяется в распределенных файловых системах (например, Sun NFS).

Директории. Логическая структура файлового архива

Количество файлов на компьютере может быть большим. Отдельные системы хранят тысячи файлов, занимающие сотни гигабайтов дискового пространства. Эффективное управление этими данными подразумевает наличие в них четкой логической структуры. Все современные файловые системы поддерживают многоуровневое именование файлов за счет наличия во внешней памяти дополнительных файлов со специальной структурой – *каталогов* (или директорий).

Каждый каталог содержит список каталогов и/или файлов, содержащихся в данном каталоге. Каталоги имеют один и тот же внутренний формат, где каждому файлу соответствует одна запись в файле директории (см., например, рис. 11.3).

Число директорий зависит от системы. В ранних ОС имелась только одна корневая директория, затем появились директории для пользователей (по одной директории на пользователя). В современных ОС используется произвольная структура дерева директорий.

Имя файла (каталога)	Тип файла (обычный или каталог)	
Anti	K	атрибуты
Games	K	атрибуты
Autoexec.bat	O	атрибуты
mouse.com	O	атрибуты

Рис. 11.3. Директории

Таким образом, файлы на диске образуют иерархическую древовидную структуру (см. рис. 11.4).

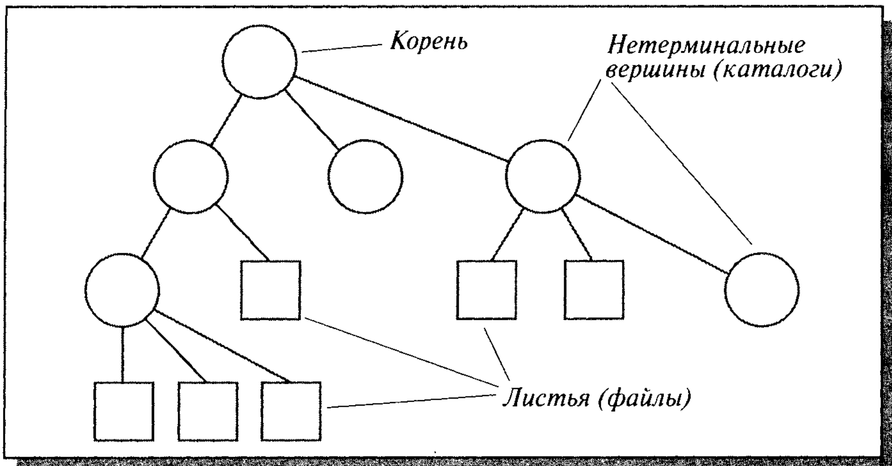


Рис. 11.4. Древовидная структура файловой системы

Существует несколько эквивалентных способов изображения дерева. Структура перевернутого дерева, приведенного на рис. 11.4, наиболее распространена. Верхнюю вершину называют корнем. Если элемент дерева не может иметь потомков, он называется терминальной вершиной или листом (в данном случае является файлом). Нелистовые вершины – справочники или каталоги — содержат списки листовых и нелистовых вершин. Путь от корня к файлу *однозначно* определяет файл.

Подобные древовидные структуры являются графами, не имеющими циклов. Можно считать, что ребра графа направлены вниз, а корень – вершина, не имеющая входящих ребер. Как мы увидим в следующей лекции, связывание файлов, которое практикуется в ряде операционных систем, приводит к образованию циклов в графе.

Внутри одного каталога имена листовых файлов уникальны. Имена файлов, находящихся в разных каталогах, могут совпадать. Для того чтобы однозначно определить файл по его имени (избежать коллизии имен), принято именовать файл так называемым *абсолютным или полным именем* (*pathname*), состоящим из списка имен вложенных каталогов, по которому можно найти путь от корня к файлу плюс имя файла в каталоге, непосредственно содержащем данный файл. То есть полное имя включает цепочку имен – путь к файлу, например `/usr/games/doom`. Такие имена уникальны. Компоненты пути разделяют различными символами: «/» (слэш) в Unix или обратными слэшем в MS-DOS (в Multics – «>»). Таким образом, использование древовидных каталогов минимизирует сложность назначения уникальных имен.

Указывать полное имя не всегда удобно, поэтому применяют другой способ задания имени – *относительный* путь к файлу. Он использует концепцию рабочей или текущей директории, которая обычно входит в состав атрибутов процесса, работающего с данным файлом. Тогда на файлы в такой директории можно ссылаться только по имени, при этом поиск файла будет осуществляться в рабочем каталоге. Это удобнее, но, по существу, то же самое, что и абсолютная форма.

Для получения доступа к файлу и локализации его блоков система должна выполнить *навигацию* по каталогам. Рассмотрим для примера путь `/usr/linux/progr.c`. Алгоритм одинаков для всех иерархических систем. Сначала в фиксированном месте на диске находится корневая директория. Затем находится компонент пути usr, т. е. в корневой директории ищется файл `/usr`. Исследуя этот файл, система понимает, что данный файл является каталогом, и блоки его данных рассматривает как список файлов и ищет следующий компонент linux в нем. Из строки для linux находится файл, соответствующий компоненту `usr/linux/`. Затем находится компонент `progr.c`, который открывается, заносится в таблицу открытых файлов и сохраняется в ней до закрытия файла.

Отклонение от типовой обработки компонентов pathname может возникнуть в том случае, когда этот компонент является не обычным каталогом с соответствующим ему индексным узлом и списком файлов, а служит точкой связывания (принято говорить «точкой монтирования») двух файловых архивов. Этот случай рассмотрен в следующей лекции.

Многие прикладные программы работают с файлами, находящимися в текущей директории, не указывая явным образом ее имени. Это дает пользователю возможность произвольным образом именовать каталоги, содержащие различные программные пакеты. Для реализации этой возможности в большинстве ОС, поддерживающих иерархическую структуру директорий, используется обозначение «.» – для текущей директории и «..» – для родительской.

Разделы диска. Организация доступа к архиву файлов

Задание пути к файлу в файловых системах некоторых ОС отличается тем, с чего начинается эта цепочка имен.

В современных ОС принято разбивать диски на *логические диски* (это низкоуровневая операция), иногда называемые *разделами* (partitions). Бывает, что, наоборот, объединяют несколько физических дисков в один логический диск (например, это можно сделать в ОС Windows NT). Поэтому в дальнейшем изложении мы будем игнорировать проблему физического выделения пространства для файлов и считать, что каждый раздел представляет собой отдельный (виртуальный) диск. Диск содержит иерархическую древовидную структуру, состоящую из набора файлов, каждый из которых является хранилищем данных пользователя, и каталогов или директорий (то есть файлов, которые содержат перечень других файлов, входящих в состав каталога), необходимых для хранения информации о файлах системы.

В некоторых системах управления файлами требуется, чтобы каждый архив файлов целиком располагался на одном диске (разделе диска). В этом случае полное имя файла начинается с имени дискового устройства, на котором установлен соответствующий диск (буквы диска). Например, `c:\util\nu\ndd.exe`. Такой способ именования используется в файловых системах DEC и Microsoft.

В других системах (Multics) вся совокупность файлов и каталогов представляет собой единое дерево. Сама система, выполняя поиск файлов по имени, начиная с корня, требовала установки необходимых дисков.

В ОС Unix предполагается наличие нескольких архивов файлов, каждый на своем разделе, один из которых считается корневым. После запуска системы можно *«смонтировать»* корневую файловую систему и ряд изолированных файловых систем в одну общую файловую систему.

Технически это осуществляется с помощью создания в корневой файловой системе специальных пустых каталогов (см. также следующую лекцию). Специальный системный вызов mount ОС Unix позволяет подключить к одному из этих пустых каталогов корневой каталог указанного архива файлов. После монтирования общей файловой системы именование файлов производится так же, как если бы она с самого начала была централизованной. Задачей ОС является беспрепятственный проход точки монтирования при получении доступа к файлу по цепочке имен. Если учесть, что обычно монтирование файловой системы производится при загрузке системы, пользователи ОС Unix обычно и не задумываются о происхождении общей файловой системы.

Операции над директориями

Как и в случае с файлами, система обязана обеспечить пользователя набором операций, необходимых для работы с директориями, реализованных через системные вызовы. Несмотря на то что директории – это файлы, логика работы с ними отличается от логики работы с обычными файлами и определяется природой этих объектов, предназначенных для поддержки структуры файлового архива. Совокупность системных вызовов для управления директориями зависит от особенностей конкретной ОС. Напомним, что операции над каталогами являются прерогативой ОС, то есть пользователь не может, например, выполнить запись в каталог начиная с текущей позиции. Рассмотрим в качестве примера некоторые системные вызовы, необходимые для работы с каталогами [Таненбаум, 2002]:

- Создание директории. Вновь созданная директория включает записи с именами '.' и '..', однако считается пустой.
- Удаление директории. Удалена может быть только пустая директория.
- Открытие директории для последующего чтения. Например, чтобы перечислить файлы, входящие в директорию, процесс должен открыть директорию и считать имена всех файлов, которые она включает.
- Закрытие директории после ее чтения для освобождения места во внутренних системных таблицах.
- Поиск. Данный системный вызов возвращает содержимое текущей записи в открытой директории. Вообще говоря, для этих целей может использоваться системный вызов read, но в этом случае от программиста потребуется знание внутренней структуры директории.
- Получение списка файлов в каталоге.
- Переименование. Имена директорий можно менять, как и имена файлов.
- Создание файла. При создании нового файла необходимо добавить в каталог соответствующий элемент.
- Удаление файла. Удаление из каталога соответствующего элемента. Если удаляемый файл присутствует только в одной директории, то он вообще удаляется из файловой системы, в противном случае система ограничивается только удалением специфицируемой записи.

Очевидно, что создание и удаление файлов предполагает также выполнение соответствующих файловых операций. Имеется еще ряд других системных вызовов, например, связанных с защитой информации.

Защита файлов

Общие проблемы безопасности ОС рассмотрены в лекциях 15–16. Информация в компьютерной системе должна быть защищена как от фи-

зического *разрушения* (reliability), так и от несанкционированного *доступа* (protection).

Здесь мы коснемся отдельных аспектов защиты, связанных с контролем доступа к файлам.

Контроль доступа к файлам

Наличие в системе многих пользователей предполагает организацию контролируемого доступа к файлам. Выполнение любой операции над файлом должно быть разрешено только в случае наличия у пользователя соответствующих привилегий. Обычно контролируются следующие операции: чтение, запись и выполнение. Другие операции, например копирование файлов или их переименование, также могут контролироваться. Однако они чаще реализуются через перечисленные. Так, операцию копирования файлов можно представить как операцию чтения и последующую операцию записи.

Списки прав доступа

Наиболее общий подход к защите файлов от несанкционированного использования – сделать доступ зависящим от идентификатора пользователя, то есть связать с каждым файлом или директорией *список прав доступа* (access control list), где перечислены имена пользователей и типы разрешенных для них способов доступа к файлу. Любой запрос на выполнение операции сверяется с таким списком. Основная проблема реализации данного способа – список может быть длинным. Чтобы разрешить всем пользователям читать файл, необходимо всех их внести в список. У такой техники есть два нежелательных следствия:

- Конструирование подобного списка может оказаться сложной задачей, особенно если мы не знаем заранее пользователей системы.
- Запись в директории должна иметь переменный размер (включать список потенциальных пользователей).

Для решения этих проблем создают классификации пользователей, например, в ОС Unix все пользователи разделены на три группы:

- *Владелец* (Owner).
- *Группа* (Group). Набор пользователей, разделяющих файл и нуждающихся в типовом способе доступа к нему.
- *Остальные* (Univers).

Это позволяет реализовать конденсированную версию списка прав доступа. В рамках такой ограниченной классификации задаются только три поля (по одному для каждой группы) для каждой контролируемой операции. В итоге в Unix операции чтения, записи и исполнения контролируются при помощи 9 бит (rwxrwxrwx).

Заключение

Итак, файловая система представляет собой набор файлов, директорий и операций над ними. Имена, структуры файлов, способы доступа к ним и их атрибуты – важные аспекты организации файловой системы. Обычно файл представляет собой неструктурированную последовательность байтов. Главная задача файловой системы – связать символьное имя файла с данными на диске. Большинство современных ОС поддерживает иерархическую систему каталогов или директорий с возможным вложением директорий. Безопасность файловой системы, базирующаяся на ведении списков прав доступа, – одна из важнейших концепций ОС.

Лекция 12. Реализация файловой системы

Реализация файловой системы связана с такими вопросами, как поддержка понятия логического блока диска, связывания имени файла и блоков его данных, а также с проблемами разделения файлов и управления дисковым пространством.

Ключевые слова: управление внешней памятью, способы выделения дискового пространства, таблица размещения файлов, FAT, индексный узел, i-node, битовый вектор, суперблок, монтирование, связь, линк, link, совместный доступ к файлу, журнализация, журналирование, кэширование, виртуальная файловая система.

Как уже говорилось, файловая система должна организовать эффективную работу с данными, хранящимися во внешней памяти, и предоставить пользователю возможности для запоминания и выборки этих данных.

Для организации хранения информации на диске пользователь вначале обычно выполняет его форматирование, выделяя на нем место для структур данных, которые описывают состояние файловой системы в целом. Затем пользователь создает нужную ему структуру каталогов (или директорий), которые, по существу, являются списками вложенных каталогов и собственно файлов. И наконец, он заполняет дисковое пространство файлами, приписывая их тому или иному каталогу. Таким образом, ОС должна предоставить в распоряжение пользователя совокупность системных вызовов, которые обеспечивают его необходимыми сервисами.

Кроме того, файловые службы могут решать проблемы проверки и сохранения целостности файловой системы, проблемы повышения производительности и ряд других.

Общая структура файловой системы

Система хранения данных на дисках может быть структурирована следующим образом (см. рис. 12.1).

Нижний уровень – оборудование. Это в первую очередь магнитные диски с подвижными головками – основные устройства внешней памяти, представляющие собой пакеты магнитных пластин (поверхностей), между которыми на одном рычаге двигается пакет магнитных головок. Шаг движения пакета головок является дискретным, и каждому положению пакета головок логически соответствует цилиндр магнитного диска. Цилиндры делятся на дорожки (треки), а каждая дорожка размечается на одно и то же количество блоков (секторов) таким образом, что в каждый блок можно

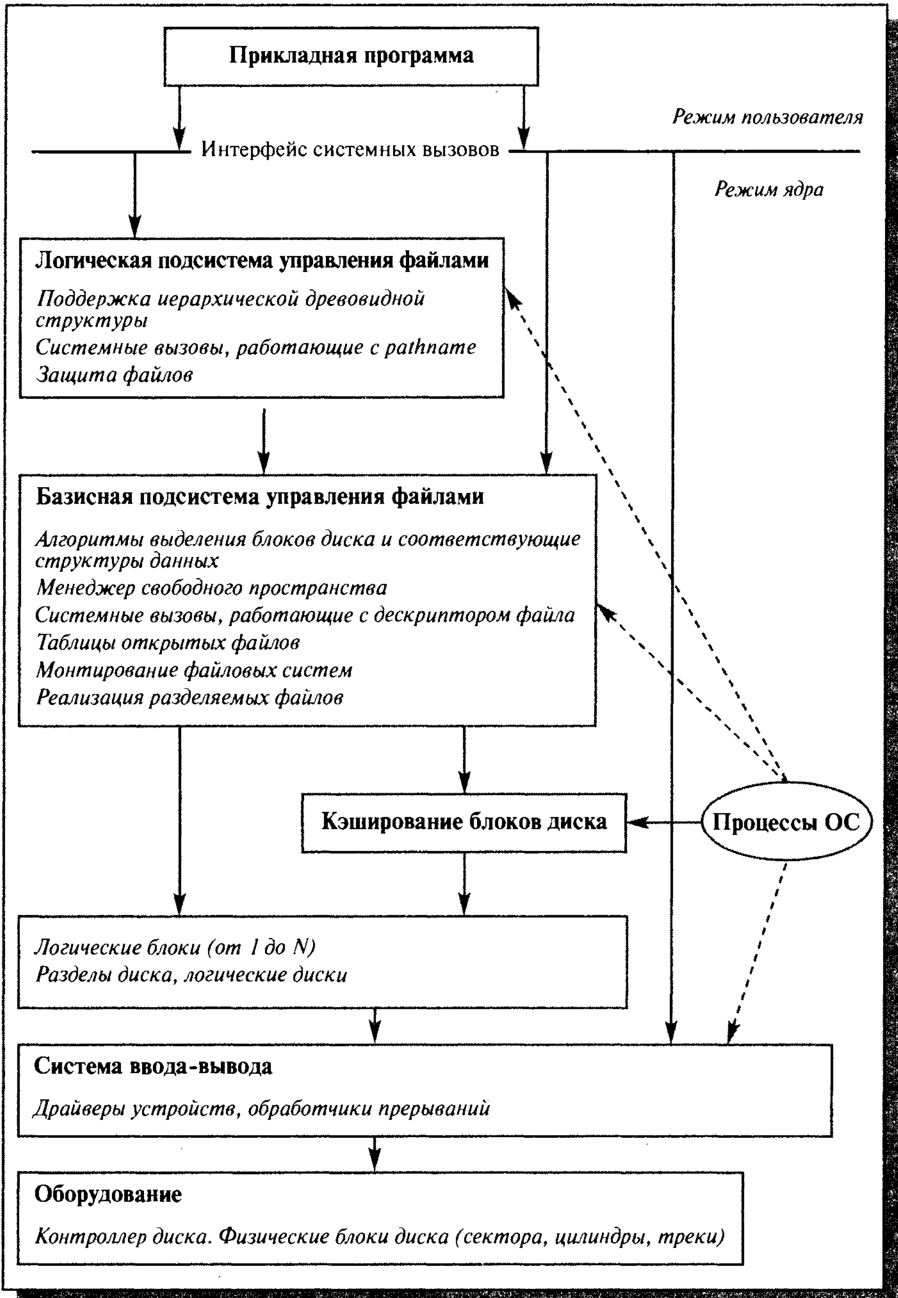


Рис. 12.1. Блок-схема файловой системы

записать по максимуму одно и то же число байтов. Следовательно, для обмена с магнитным диском на уровне аппаратуры нужно указать номер цилиндра, номер поверхности, номер блока на соответствующей дорожке и число байтов, которое нужно записать или прочитать от начала этого блока. Таким образом, диски могут быть разбиты на блоки фиксированного размера и можно непосредственно получить доступ к любому блоку (организовать прямой доступ к файлам).

Непосредственно с устройствами (дисками) взаимодействует часть ОС, называемая *системой ввода-вывода* (см. лекцию 13). Система ввода-вывода предоставляет в распоряжение более высокоуровневого компонента ОС – файловой системы – используемое дисковое пространство в виде *непрерывной последовательности блоков фиксированного размера*. Система ввода-вывода имеет дело с *физическими* блоками диска, которые характеризуются адресом, например диск 2, цилиндр 75, сектор 11. Файловая система имеет дело с *логическими* блоками, каждый из которых имеет номер (от 0 или 1 до N). Размер логических блоков файла совпадает или является кратным размеру физического блока диска и может быть задан равным размеру страницы виртуальной памяти, поддерживаемой аппаратурой компьютера совместно с операционной системой.

В структуре системы управления файлами можно выделить *базисную* подсистему, которая отвечает за выделение дискового пространства конкретным файлам, и более высокоуровневую *логическую* подсистему, которая использует структуру дерева директорий для предоставления модулю базисной подсистемы необходимой ей информации, исходя из символического имени файла. Она также ответственна за авторизацию доступа к файлам (см. лекции 11 и 16).

Стандартный запрос на открытие (open) или создание (creat) файла поступает от прикладной программы к логической подсистеме. Логическая подсистема, используя структуру директорий, проверяет права доступа и вызывает базовую подсистему для получения доступа к блокам файла. После этого файл считается открытым, он содержится в таблице открытых файлов, и прикладная программа получает в свое распоряжение дескриптор (или handle в системах Microsoft) этого файла. Дескриптор файла является ссылкой на файл в таблице открытых файлов и используется в запросах прикладной программы на чтение-запись из этого файла. Запись в таблице открытых файлов указывает через систему выделения блоков диска на блоки данного файла. Если к моменту открытия файл уже используется другим процессом, то есть содержится в таблице открытых файлов, то после проверки прав доступа к файлу может быть организован совместный доступ. При этом новому процессу также возвращается дескриптор – ссылка на файл в таблице открытых файлов. Далее в тексте подробно проанализирована работа наиболее важных системных вызовов.

Управление внешней памятью

Прежде чем описывать структуру данных файловой системы на диске, необходимо рассмотреть алгоритмы выделения дискового пространства и способы учета свободной и занятой дисковой памяти. Эти задачи связаны между собой.

Методы выделения дискового пространства

Ключевым, безусловно, является вопрос, какой тип структур используется для учета отдельных блоков файла, то есть способ связывания файлов с блоками диска. В ОС используется несколько методов выделения файлу дискового пространства. Для каждого из методов запись в директории, соответствующая символьному имени файла, содержит указатель, следуя которому можно найти все блоки данного файла.

Выделение непрерывной последовательностью блоков

Простейший способ – хранить каждый файл как непрерывную последовательность блоков диска. При непрерывном расположении файл характеризуется адресом и длиной (в блоках). Файл, стартующий с блока b, занимает затем блоки b + 1, b + 2, ... b + n – 1.

Эта схема имеет два преимущества. Во-первых, ее легко реализовать, так как выяснение местонахождения файла сводится к вопросу, где находится первый блок. Во-вторых, она обеспечивает хорошую производительность, так как целый файл может быть считан за одну дисковую операцию.

Непрерывное выделение используется в ОС IBM/CMS, в ОС RSX-11 (для выполняемых файлов) и в ряде других.

Этот способ распространен мало, и вот почему. В процессе эксплуатации диск представляет собой некоторую совокупность свободных и занятых фрагментов. Не всегда имеется подходящий по размеру свободный фрагмент для нового файла. Проблема непрерывного расположения может рассматриваться как частный случай более общей проблемы выделения блока нужного размера из списка свободных блоков. Типовыми решениями этой задачи являются стратегии первого подходящего, наиболее подходящего и наименее подходящего (сравните с проблемой выделения памяти в методе с динамическим распределением). Как и в случае выделения нужного объема оперативной памяти в схеме с динамическими разделами (см. лекцию 8), метод страдает от ***внешней фрагментации***, в большей или меньшей степени, в зависимости от размера диска и среднего размера файла.

Кроме того, непрерывное распределение внешней памяти неприменимо до тех пор, пока неизвестен максимальный размер файла. Ино-

гда размер выходного файла оценить легко (при копировании). Чаще, однако, это трудно сделать, особенно в тех случаях, когда размер файла меняется. Если места не хватило, то пользовательская программа может быть приостановлена с учетом выделения дополнительного места для файла при последующем рестарте. Некоторые ОС используют модифицированный вариант непрерывного выделения – основные блоки файла + резервные блоки. Однако с выделением блоков из резерва возникают те же проблемы, так как приходится решать задачу выделения непрерывной последовательности блоков диска теперь уже из совокупности резервных блоков.

Единственным приемлемым решением перечисленных проблем является периодическое уплотнение содержимого внешней памяти, или «сборка мусора», цель которой состоит в объединении свободных участков в один большой блок. Но это дорогостоящая операция, которую невозможно осуществлять слишком часто.

Таким образом, когда содержимое диска постоянно изменяется, данный метод нерационален. Однако для *стационарных* файловых систем, например для файловых систем компакт-дисков, он вполне пригоден.

Связный список

Внешняя фрагментация – основная проблема рассмотренного выше метода – может быть устранена за счет представления файла в виде связного списка блоков диска. Запись в директории содержит указатель на первый и последний блоки файла (иногда в качестве варианта используется специальный знак конца файла – EOF). Каждый блок содержит указатель на следующий блок (см. рис. 12.2).

Внешняя фрагментация для данного метода отсутствует. Любой свободный блок может быть использован для удовлетворения запроса. Заме-

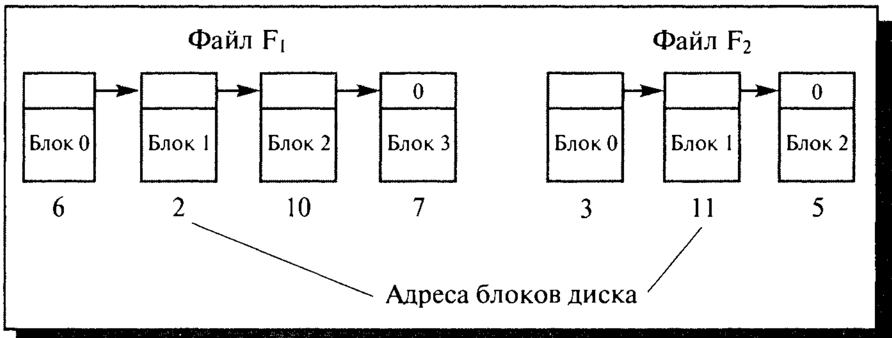


Рис. 12.2. Хранение файла в виде связного списка дисковых блоков

тим, что нет необходимости декларировать размер файла в момент создания. Файл может расти неограниченно.

Связное выделение имеет, однако, несколько существенных недостатков.

Во-первых, при прямом доступе к файлу для поиска i-го блока нужно осуществить несколько обращений к диску, последовательно считывая блоки от 1 до i – 1, то есть выборка логически смежных записей, которые занимают физически несмежные секторы, может требовать много времени. Здесь мы теряем все преимущества прямого доступа к файлу.

Во-вторых, данный способ не очень надежен. Наличие дефектного блока в списке приводит к потере информации в оставшейся части файла и потенциально к потере дискового пространства, отведенного под этот файл.

Наконец, для указателя на следующий блок внутри блока нужно выделить место, что не всегда удобно. Емкость блока, традиционно являющаяся степенью двойки (многие программы читают и пишут блоками по степеням двойки), таким образом, перестает быть степенью двойки, так как указатель отбирает несколько байтов.

Поэтому метод связного списка обычно в чистом виде не используется.

Таблица отображения файлов

Одним из вариантов предыдущего способа является хранение указателей не в дисковых блоках, а в индексной таблице в памяти, которая называется таблицей отображения файлов (FAT – File Allocation Table) (см. рис. 12.3). Этой схемы придерживаются многие ОС (MS-DOS, OS/2, MS Windows и др.).

По-прежнему существенно, что запись в директории содержит только ссылку на первый блок. Далее при помощи таблицы FAT можно локализовать блоки файла независимо от его размера. В тех строках таблицы, которые соответствуют последним блокам файлов, обычно записывается некоторое граничное значение, например EOF.

Главное достоинство данного подхода состоит в том, что по таблице отображения можно судить о физическом соседстве блоков, располагающихся на диске, и при выделении нового блока можно легко найти свободный блок диска, находящийся поблизости от других блоков данного файла. Минусом данной схемы может быть необходимость хранения в памяти этой довольно большой таблицы.

Индексные узлы

Наиболее распространенный метод выделения файлу блоков диска – связать с каждым файлом небольшую таблицу, называемую индексным уз-

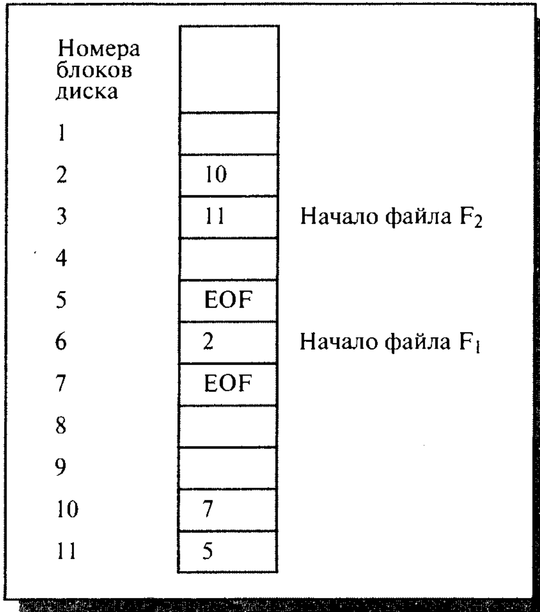


Рис. 12.3. Метод связного списка с использованием таблицы в оперативной памяти

лом (i-node), которая перечисляет атрибуты и дисковые адреса блоков файла (см. рис. 12.4). Запись в директории, относящаяся к файлу, содержит адрес индексного блока. По мере заполнения файла указатели на блоки диска в индексном узле принимают осмысленные значения.

Индексирование поддерживает прямой доступ к файлу, без ущерба от внешней фрагментации. Индексированное размещение широко распространено и поддерживает как последовательный, так и прямой доступ к файлу.

Обычно применяется комбинация одноуровневого и многоуровневых индексов. Первые несколько адресов блоков файла хранятся непосредственно в индексном узле. Таким образом, для маленьких файлов индексный узел хранит всю необходимую информацию об адресах блоков диска. Для больших файлов один из адресов индексного узла указывает на блок косвенной адресации. Данный блок содержит адреса дополнительных блоков диска. Если этого недостаточно, используется блок двойной косвенной адресации, который содержит адреса блоков косвенной адресации. Если и этого не хватает, используется блок тройной косвенной адресации.

Данную схему используют файловые системы Unix (а также файловые системы HPFS, NTFS и др.). Такой подход позволяет при фиксиро-

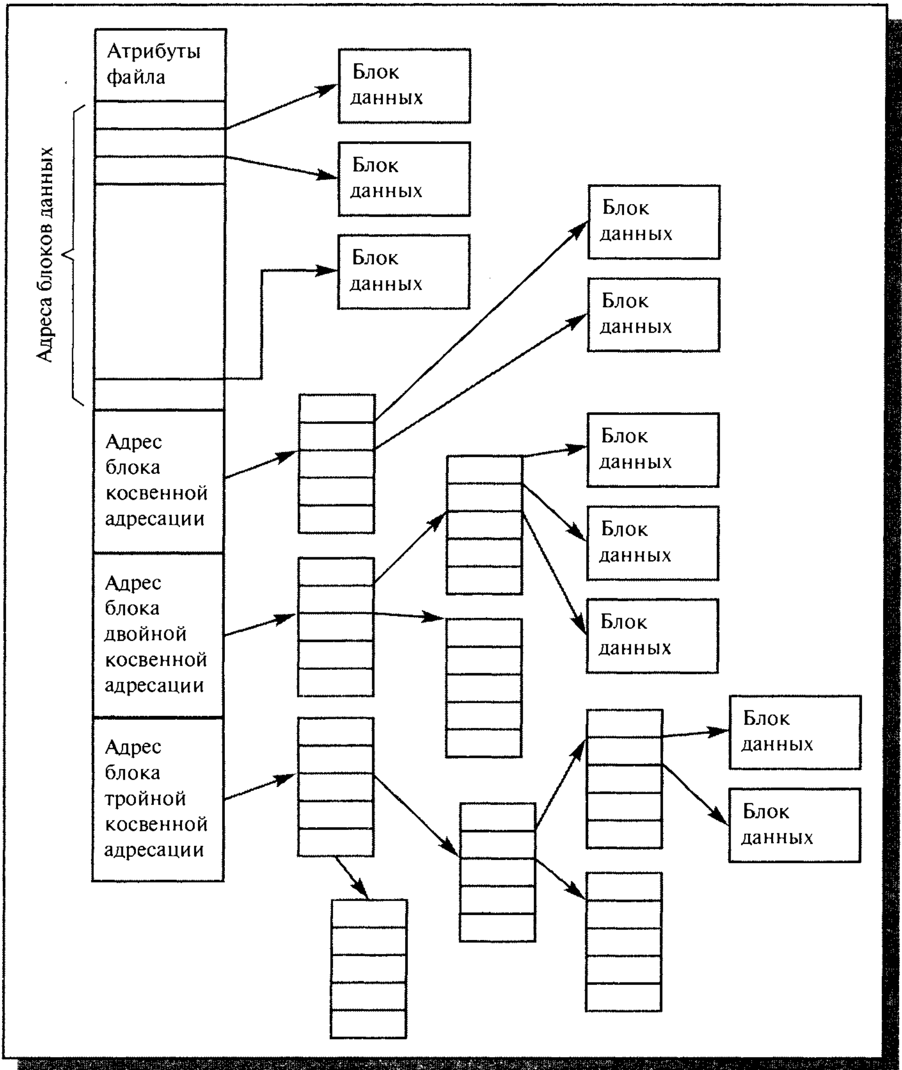


Рис. 12.4. Структура индексного узла

ванном, относительно небольшом размере индексного узла поддерживать работу с файлами, размер которых может меняться от нескольких байтов до нескольких гигабайтов. Существенно, что для маленьких файлов используется только прямая адресация, обеспечивающая максимальную производительность.

Управление свободным и занятым дисковым пространством

Дисковое пространство, не выделенное ни одному файлу, также должно быть управляемым. В современных ОС используется несколько способов учета используемого места на диске. Рассмотрим наиболее распространенные.

Учет при помощи организации битового вектора

Часто список свободных блоков диска реализован в виде битового вектора (bit map или bit vector). Каждый блок представлен одним битом, принимающим значение 0 или 1, в зависимости от того, занят он или свободен. Например, 00111100111100011000001 ... .

Главное преимущество этого подхода состоит в том, что он относительно прост и эффективен при нахождении первого свободного блока или n последовательных блоков на диске. Многие компьютеры имеют инструкции манипулирования битами, которые могут использоваться для этой цели. Например, компьютеры семейств Intel и Motorola имеют инструкции, при помощи которых можно легко локализовать первый единичный бит в слове.

Описываемый метод учета свободных блоков используется в Apple Macintosh.

Несмотря на то что размер описанного битового вектора является наименьшим из всех возможных структур, даже такой вектор может оказаться довольно большого размера. Поэтому данный метод эффективен, только если битовый вектор помещается в памяти целиком, что возможно лишь для относительно небольших дисков. Например, диск размером 4 Гбайт с блоками по 4 Кбайт нуждается в таблице размером 128 Кбайт для управления свободными блоками. Иногда, если битовый вектор становится слишком большим, для ускорения поиска в нем его разбивают на регионы и организуют резюмирующие структуры данных, содержащие сведения о количестве свободных блоков для каждого региона.

Учет при помощи организации связного списка

Другой подход – связать в список все свободные блоки, размещая указатель на первый свободный блок в специально отведенном месте диска, попутно кэшируя в памяти эту информацию.

Подобная схема не всегда эффективна. Для трассирования списка нужно выполнить много обращений к диску. Однако, к счастью, нам необходим, как правило, только первый свободный блок.

Иногда прибегают к модификации подхода связного списка, организуя хранение адресов n свободных блоков в первом свободном блоке. Первые n – 1 этих блоков действительно используются. Последний блок содержит адреса других n блоков и т. д.

Существуют и другие методы, например, свободное пространство можно рассматривать как файл и вести для него соответствующий индексный узел.

Размер блока

Размер логического блока играет важную роль. В некоторых системах (Unix) он может быть задан при форматировании диска. Небольшой размер блока будет приводить к тому, что каждый файл будет содержать много блоков. Чтение блока осуществляется с задержками на поиск и вращение, таким образом, файл из многих блоков будет читаться медленно. Большие блоки обеспечивают более высокую скорость обмена с диском, но из-за внутренней фрагментации (каждый файл занимает целое число блоков, и в среднем половина последнего блока пропадает) снижается процент полезного дискового пространства.

Для систем со страничной организацией памяти характерна сходная проблема с размером страницы.

Проведенные исследования показали, что большинство файлов имеют небольшой размер. Например, в Unix приблизительно 85% файлов имеют размер менее 8 Кбайт и 48% – менее 1 Кбайта.

Можно также учесть, что в системах с виртуальной памятью желательно, чтобы единицей пересылки диск–память была страница (наиболее распространенный размер страниц памяти – 4 Кбайта). Отсюда обычный компромиссный выбор блока размером 512 байт, 1 Кбайт, 2 Кбайт, 4 Кбайт.

Структура файловой системы на диске

Рассмотрение методов работы с дисковым пространством дает общее представление о совокупности служебных данных, необходимых для описания файловой системы. Структура служебных данных типовой файловой системы, например Unix, на одном из разделов диска, таким образом, может состоять из четырех основных частей (см. рис. 12.5).

В начале раздела находится ***суперблок***, содержащий общее описание файловой системы, например:

- тип файловой системы;
- размер файловой системы в блоках;
- размер массива индексных узлов;
- размер логического блока.

Суперблок	Структуры данных, описывающие свободное дисковое пространство и свободные индексные узлы	Массив индексных узлов	Блоки диска данных файлов

Рис. 12.5. Примерная структура файловой системы на диске

Описанные структуры данных создаются на диске в результате его *форматирования* (например, утилитами format, makefs и др.). Их наличие позволяет обращаться к данным на диске как к файловой системе, а не как к обычной последовательности блоков.

В файловых системах современных ОС для повышения устойчивости поддерживается несколько копий суперблока. В некоторых версиях Unix суперблок включал также и структуры данных, управляющие распределением дискового пространства, в результате чего суперблок непрерывно подвергался модификации, что снижало надежность файловой системы в целом. Выделение структур данных, описывающих дисковое пространство, в отдельную часть является более правильным решением.

Массив индексных узлов (ilist) содержит список индексов, соответствующих файлам данной файловой системы. Размер массива индексных узлов определяется администратором при установке системы. Максимальное число файлов, которые могут быть созданы в файловой системе, определяется числом доступных индексных узлов.

В блоках данных хранятся реальные данные файлов. Размер логического блока данных может задаваться при форматировании файловой системы. Заполнение диска содержательной информацией предполагает использование блоков хранения данных для файлов директорий и обычных файлов и имеет следствием модификацию массива индексных узлов и данных, описывающих пространство диска. Отдельно взятый блок данных может принадлежать одному и только одному файлу в файловой системе.

Реализация директорий

Как уже говорилось, директория или каталог – это файл, имеющий вид таблицы и хранящий список входящих в него файлов или каталогов. Основная задача файлов-директорий – поддержка иерархической древовидной структуры файловой системы. Запись в директории имеет определенный для данной ОС формат, зачастую неизвестный пользователю, поэтому блоки данных файла-директории заполняются не через операции

записи, а при помощи специальных системных вызовов (например, создание файла).

Для доступа к файлу ОС использует путь (pathname), сообщенный пользователем. Запись в директории связывает имя файла или имя поддиректории с блоками данных на диске (см. рис. 12.6). В зависимости от способа выделения файлу блоков диска (см. раздел «Методы выделения дискового пространства») эта ссылка может быть номером первого блока или номером индексного узла. В любом случае обеспечивается связь символьного имени файла с данными на диске.

Когда система открывает файл, она ищет его имя в директории. Затем из записи в директории или из структуры, на которую запись в директории указывает, извлекаются атрибуты и адреса блоков файла на диске. Эта информация помещается в системную таблицу в главной памяти. Все последующие ссылки на данный файл используют эту информацию. Атрибуты файла можно хранить непосредственно в записи в директории, как показано на рис. 12.6. Однако для организации совместного доступа к файлам удобнее хранить атрибуты в индексном узле, как это делается в Unix.

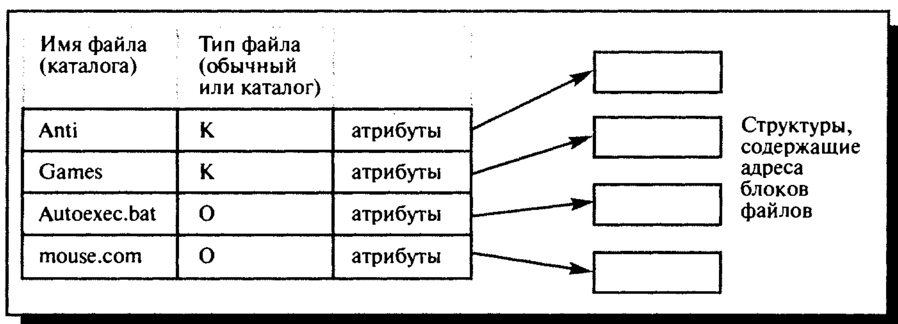


Рис. 12.6. Реализация директорий

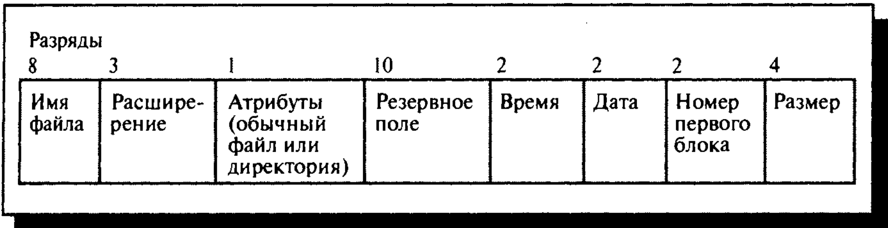
Рассмотрим несколько конкретных примеров.

Примеры реализации директорий в некоторых ОС

Директории в ОС MS-DOS

В ОС MS-DOS типовая запись в директории имеет вид, показанный на рис. 12.7.

В ОС MS-DOS, как и в большинстве современных ОС, директории могут содержать поддиректории (специфицируемые битом атрибута), что позволяет конструировать произвольное дерево директорий файловой системы.



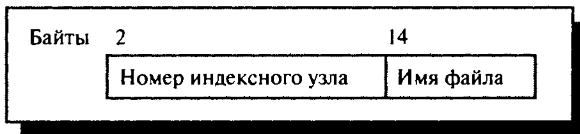
Разряды 8	3	1	10	2	2	2	4
Имя файла	Расшире-рение	Атрибуты (обычный файл или директория)	Резервное поле	Время	Дата	Номер первого блока	Размер

Рис. 12.7. Вариант записи в директории MS-DOS

Номер первого блока используется в качестве индекса в таблице FAT. Далее по цепочке в этой таблице могут быть найдены остальные блоки.

Директории в ОС Unix

Структура директории проста. Каждая запись содержит имя файла и номер его индексного узла (см. рис. 12.8). Вся остальная информация о файле (тип, размер, время модификации, владелец и т. д. и номера дисковых блоков) находится в индексном узле.



Байты 2	14
Номер индексного узла	Имя файла

Рис. 12.8. Вариант записи в директории Unix

В более поздних версиях Unix форма записи претерпела ряд изменений, например имя файла описывается структурой. Однако суть осталась прежней.

Поиск в директории

Список файлов в директории обычно не является упорядоченным по именам файлов. Поэтому правильный выбор алгоритма поиска имени файла в директории имеет большое влияние на эффективность и надежность файловых систем.

Линейный поиск

Существует несколько стратегий просмотра списка символьных имен. Простейшей из них является линейный поиск. Директория просматривается с самого начала, пока не встретится нужное имя файла. Хотя это наименее эффективный способ поиска, оказывается, что в боль-

шинстве случаев он работает с приемлемой производительностью. Например, авторы Unix утверждали, что линейного поиска вполне достаточно. По-видимому, это связано с тем, что на фоне относительно медленного доступа к диску некоторые задержки, возникающие в процессе сканирования списка, несущественны.

Метод прост, но требует временных затрат. Для создания нового файла вначале нужно проверить директорию на наличие такого же имени. Затем имя нового файла вставляется в конец директории (если, разумеется, файл с таким же именем в директории не существует; в противном случае нужно информировать пользователя). Для удаления файла нужно также выполнить поиск его имени в списке и пометить запись как неиспользуемую.

Реальный недостаток данного метода – последовательный поиск файла. Информация о структуре директории используется часто, и неэффективный способ поиска будет заметен пользователями. Можно свести поиск к бинарному, если отсортировать список файлов. Однако это усложнит создание и удаление файлов, так как требуется перемещение большого объема информации.

Хеш-таблица

Хеширование (см., например, [Ахо, 2001]) – другой способ, который может использоваться для размещения и последующего поиска имени файла в директории. В данном методе имена файлов также хранятся в каталоге в виде линейного списка, но дополнительно используется хеш-таблица. Хеш-таблица, точнее построенная на ее основе хеш-функция, позволяет по имени файла получить указатель на имя файла в списке. Таким образом, можно существенно уменьшить время поиска.

В результате хеширования могут возникать коллизии, то есть ситуации, когда функция хеширования, примененная к разным именам файлов, дает один и тот же результат. Обычно имена таких файлов объединяют в связные списки, предполагая в дальнейшем осуществление в них последовательного поиска нужного имени файла. Выбор подходящего алгоритма хеширования позволяет свести к минимуму число коллизий. Однако всегда есть вероятность неблагоприятного исхода, когда непропорционально большому числу имен файлов функция хеширования ставит в соответствие один и тот же результат. В таком случае преимущество использования этой схемы по сравнению с последовательным поиском практически утрачивается.

Другие методы поиска

Помимо описанных методов поиска имени файла, в директории существуют и другие. В качестве примера можно привести организацию поиска в каталогах файловой системы NTFS при помощи так называемого В-дерева, которое стало стандартным способом организации индексов в системах баз данных (см. [Ахо, 2001]).

Монтирование файловых систем

Так же как файл должен быть открыт перед использованием, так и файловая система, хранящаяся на разделе диска, должна быть смонтирована, чтобы стать доступной процессам системы.

Функция `mount` (монтировать) связывает файловую систему из указанного раздела на диске с существующей иерархией файловых систем, а функция `umount` (демонтировать) выключает файловую систему из иерархии. Функция `mount`, таким образом, дает пользователям возможность обращаться к данным в дисковом разделе как к файловой системе, а не как к последовательности дисковых блоков.

Процедура монтирования состоит в следующем. Пользователь (в Unix это суперпользователь) сообщает ОС имя устройства и место в файловой структуре (имя пустого каталога), куда нужно присоединить файловую систему (точка монтирования) (см. рис. 12.9 и рис. 12.10). Например, в ОС Unix библиотечный вызов `mount` имеет вид

```
mount(special pathname,directory pathname,options);
```

где `special pathname` – имя специального файла устройства (в общем случае имя раздела), соответствующего дисковому разделу с монтируемой файловой системой, `directory pathname` – каталог в существующей иерархии, где будет монтироваться файловая система (другими словами, точка или место монтирования), а `options` указывает, следует ли монтировать файловую систему «только для чтения» (при этом не будут выполняться такие функции, как `write` и `creat`, которые производят запись в файловую систему). Затем ОС должна убедиться, что устройство содержит действительную файловую систему ожидаемого формата с суперблоком, списком индексов и корневым индексом.

Некоторые ОС осуществляют монтирование автоматически, как только встретят диск в первый раз (жесткие диски на этапе загрузки, гибкие – когда они вставлены в дисковод), ОС ищет файловую систему на устройстве. Если файловая система на устройстве имеется, она монтируется на корневом уровне, при этом к цепочке имен абсолютного имени файла (pathname) добавляется буква раздела.

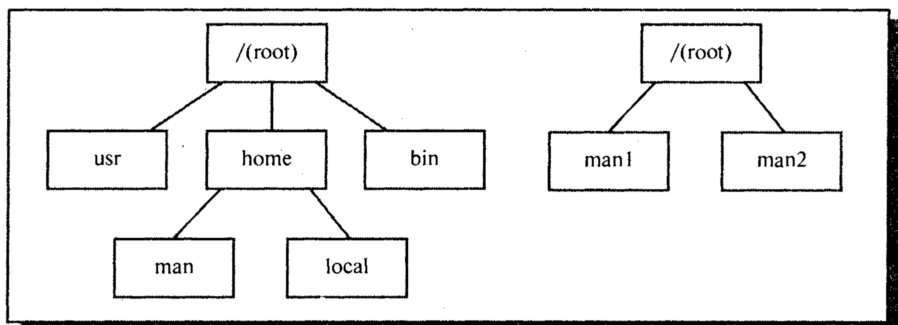


Рис. 12.9. Две файловые системы до монтирования

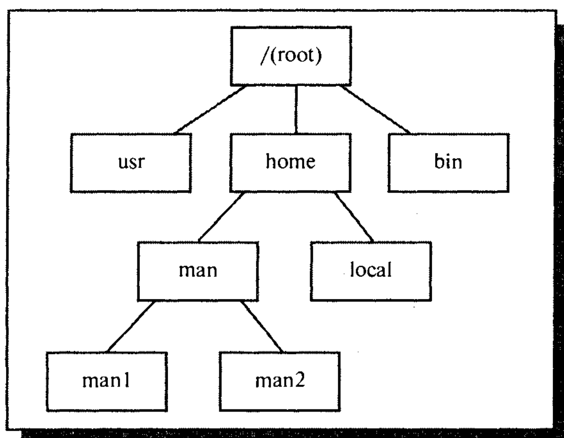


Рис. 12.10. Общая файловая система после монтирования

Ядро поддерживает таблицу монтирования с записями о каждой смонтированной файловой системе. В каждой записи содержится информация о вновь смонтированном устройстве, о его суперблоке и корневом каталоге, а также сведения о точке монтирования. Для устранения потенциально опасных побочных эффектов число линков (см. следующий раздел) к каталогу – точке монтирования – должно быть равно 1. Занесение информации в таблицу монтирования производится немедленно, поскольку может возникнуть конфликт между двумя процессами. Например, если монтирующий процесс приостановлен для открытия устройства или считывания суперблока файловой системы, а другой процесс тем временем может попытаться смонтировать файловую систему.

Наличие в логической структуре файлового архива точек монтирования требует аккуратной реализации алгоритмов, осуществляющих на-

вигацию по каталогам. Точку монтирования можно пересечь двумя способами: из файловой системы, где производится монтирование, в файловую систему, которая монтируется (в направлении от глобального корня к листу), и в обратном направлении. Алгоритмы поиска файлов должны предусматривать ситуации, в которых очередной компонент пути к файлу является точкой монтирования, когда вместо анализа индексного узла очередной директории приходится осуществлять обработку суперблока монтированной системы.

Связывание файлов

Иерархическая организация, положенная в основу древовидной структуры файловой системы современных ОС, не предусматривает выражения отношений, в которых потомки связываются более чем с одним предком. Такая негибкость частично устраняется возможностью реализации связывания файлов или организации *линков* (links).

Ядро позволяет пользователю связывать каталоги, упрощая написание программ, требующих пересечения дерева файловой системы (см. рис. 12.11). Часто имеет смысл хранить под разными именами одну и ту же команду (выполняемый файл). Например, выполняемый файл традиционного текстового редактора ОС Unix `vi` обычно может вызываться под именами `ex`, `edit`, `vi`, `view` и `vedit` файловой системы. Соединение между директорией и разделяемым файлом называется «связью» или «ссылкой» (link). Дерево файловой системы превращается в циклический граф.

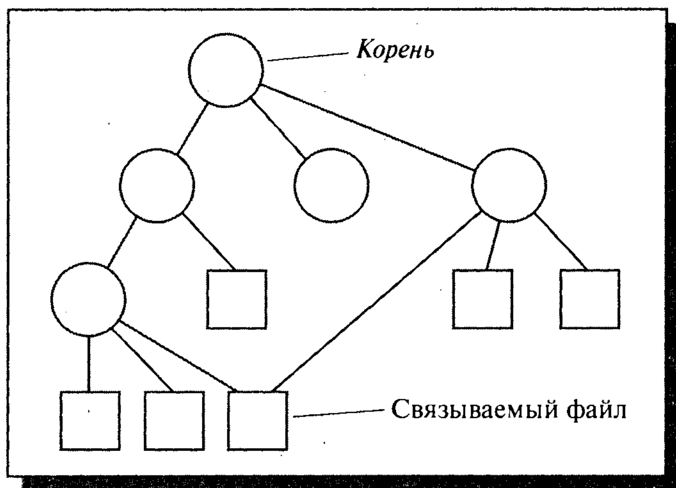


Рис. 12.11. Структура файловой системы с возможностью связывания файла с новым именем

Это удобно, но создает ряд дополнительных проблем.

Простейший способ реализовать связывание файла – просто дублировать информацию о нем в обеих директориях. При этом, однако, может возникнуть проблема совместимости в случае, если владельцы этих директорий попытаются независимо друг от друга изменить содержимое файла. Например, в ОС CP/M запись в директории о файле непосредственно содержит адреса дисковых блоков. Поэтому копии тех же дисковых адресов должны быть сделаны и в другой директории, куда файл линкуется. Если один из пользователей что-то добавляет к файлу, новые блоки будут перечислены только у него в директории и не будут «видны» другому пользователю.

Проблема такого рода может быть решена двумя способами. Первый из них – так называемая *жесткая* связь (hard link). Если блоки данных файла перечислены не в директории, а в небольшой структуре данных (например, в индексном узле), связанной собственно с файлом, то второй пользователь может связаться непосредственно с этой, уже существующей структурой.

Альтернативное решение – создание нового файла, который содержит путь к связываемому файлу. Такой подход называется *символической* линковкой (soft или symbolic link). При этом в соответствующем каталоге создается элемент, в котором имени связи сопоставляется некоторое имя файла (этот файл даже не обязан существовать к моменту создания символической связи). Для символической связи может создаваться отдельный индексный узел и даже заводиться отдельный блок данных для хранения потенциально длинного имени файла.

Каждый из этих методов имеет свои минусы. В случае жесткой связи возникает необходимость поддержки счетчика ссылок на файл для корректной реализации операции удаления файла. Например, в Unix такой счетчик является одним из атрибутов, хранящихся в индексном узле. Удаление файла одним из пользователей уменьшает количество ссылок на файл на 1. Реальное удаление файла происходит, когда число ссылок на файл становится равным 0.

В случае символической линковки такая проблема не возникает, так как только реальный владелец имеет ссылку на индексный узел файла. Если собственник удаляет файл, то он разрушается, и попытки других пользователей работать с ним закончатся провалом. Удаление символического линка на файл никак не влияет. Проблема организации символической связи – потенциальное снижение скорости доступа к файлу. Файл символического линка хранит путь к файлу, содержащий список вложенных директорий, для прохождения по которому необходимо осуществить несколько обращений к диску.

Символический линк имеет то преимущество, что он может использоваться для организации удобного доступа к файлам удаленных компьютеров, если, например, добавить к пути сетевой адрес удаленной машины.

Циклический граф – структура более гибкая, нежели простое дерево, но работа с ней требует большой аккуратности. Поскольку теперь к файлу существует несколько путей, программа поиска файла может найти его на диске несколько раз. Простейшее практическое решение данной проблемы – ограничить число директорий при поиске. Полное устранение циклов при поиске – довольно трудоемкая процедура, выполняемая специальными утилитами и связанная с многократной трассировкой директорий файловой системы.

Кооперация процессов при работе с файлами

Когда различные пользователи работают вместе над проектом, они часто нуждаются в разделении файлов.

Разделяемый файл – разделяемый ресурс. Как и в случае любого совместно используемого ресурса, процессы должны синхронизировать доступ к совместно используемым файлам, каталогам, чтобы избежать тупиковых ситуаций, дискриминации отдельных процессов и снижения производительности системы.

Например, если несколько пользователей одновременно редактируют какой-либо файл и не принято специальных мер, то результат будет непредсказуем и зависит от того, в каком порядке осуществлялись записи в файл. Между двумя операциями read одного процесса другой процесс может модифицировать данные, что для многих приложений неприемлемо. Простейшее решение данной проблемы – предоставить возможность одному из процессов захватить файл, то есть блокировать доступ к разделяемому файлу других процессов на все время, пока файл остается открытым для данного процесса. Однако это было бы недостаточно гибко и не соответствовало бы характеру поставленной задачи.

Рассмотрим вначале *грубый* подход, то есть временный захват пользовательским процессом файла или записи (части файла между указанными позициями).

Системный вызов, позволяющий установить и проверить блокировки на файл, является неотъемлемым атрибутом современных многопользовательских ОС. В принципе, было бы логично связать синхронизацию доступа к файлу как к единому целому с системным вызовом open (т. е., например, открытие файла в режиме записи или обновления могло бы означать его монопольную блокировку соответствующим процессом, а открытие в режиме чтения – совместную блокировку). Так поступают во

многих операционных системах (начиная с ОС Multics). В ОС Unix это не так, что имеет исторические причины.

В первой версии системы Unix, разработанной Томпсоном и Ричи, механизм захвата файла отсутствовал. Применялся очень простой подход к обеспечению параллельного (от нескольких процессов) доступа к файлам: система позволяла любому числу процессов одновременно открывать один и тот же файл в любом режиме (чтения, записи или обновления) и не предпринимала никаких синхронизационных действий. Вся ответственность за корректность совместной обработки файла ложилась на использующие его процессы, и система даже не предоставляла каких-либо особых средств для синхронизации доступа процессов к файлу. Однако впоследствии для того, чтобы повысить привлекательность системы для коммерческих пользователей, работающих с базами данных, в версию V системы были включены механизмы захвата файла и записи, базирующиеся на системном вызове `fcntl`.

Допускается два варианта синхронизации: с ожиданием, когда требование блокировки может привести к откладыванию процесса до того момента, когда это требование может быть удовлетворено, и без ожидания, когда процесс немедленно оповещается об удовлетворении требования блокировки или о невозможности ее удовлетворения в данный момент.

Установленные блокировки относятся только к тому процессу, который их установил, и не наследуются процессами-потомками этого процесса. Более того, даже если некоторый процесс пользуется синхронизационными возможностями системного вызова `fcntl`, другие процессы по-прежнему могут работать с тем файлом без всякой синхронизации. Другими словами, это дело группы процессов, совместно использующих файл, – договориться о способе синхронизации параллельного доступа.

Более тонкий подход заключается в прозрачной для пользователя блокировке отдельных структур ядра, отвечающих за работу с файлами части пользовательских данных. Например, в ОС Unix во время системного вызова, осуществляющего ту или иную операцию с файлом, как правило, происходит блокирование индексного узла, содержащего адреса блоков данных файла. Может показаться, что организация блокировок или запрета более чем одному процессу работать с файлом во время выполнения системного вызова является излишней, так как в подавляющем большинстве случаев выполнение системных вызовов и так не прерывается, то есть ядро работает в условиях невытесняющей многозадачности. Однако в данном случае это не совсем так. Операции чтения и записи занимают продолжительное время и лишь инициируются центральным процессором, а осуществляются по независимым каналам, поэтому установка блокировок на время системного вызова является необходимой га-

рантией атомарности операций чтения и записи. На практике оказывается достаточным заблокировать один из буферов кэша диска, в заголовке которого ведется список процессов, ожидающих освобождения данного буфера. Таким образом, в соответствии с семантикой Unix изменения, сделанные одним пользователем, немедленно становятся «видны» другому пользователю, который держит данный файл открытым одновременно с первым.

Примеры разрешения коллизий и тупиковых ситуаций

Логика работы системы в сложных ситуациях может проиллюстрировать особенности организации мультидоступа.

Рассмотрим в качестве примера *образование потенциального тупика* при создании связи (link), когда разрешен совместный доступ к файлу [Bach, 1986].

Два процесса, выполняющие одновременно следующие функции

```
процесс A:   link("a/b/c/d","e/f/g");
процесс B:   link("e/f","a/b/c/d/ee");
```

могут зайти в тупик. Предположим, что процесс A обнаружил индекс файла "a/b/c/d" в тот самый момент, когда процесс B обнаружил индекс файла "e/f". Фраза «в тот же самый момент» означает, что системой достигнуто состояние, при котором каждый процесс получил искомый индекс. Когда же теперь процесс A попытается получить индекс файла "e/f", он приостановит свое выполнение до тех пор, пока индекс файла "f" не освободится. В то же время процесс B пытается получить индекс каталога "a/b/c/d" и приостанавливается в ожидании освобождения индекса файла "d". Процесс A будет удерживать заблокированным индекс, нужный процессу B, а процесс B, в свою очередь, будет удерживать заблокированным индекс, необходимый процессу A.

Для предотвращения этого классического примера взаимной блокировки в файловой системе принято, чтобы ядро освобождало индекс исходного файла после увеличения значения счетчика связей. Тогда, поскольку первый из ресурсов (индекс) свободен при обращении к следующему ресурсу, взаимной блокировки не происходит.

Поводов для нежелательной *конкуренции* между процессами много, особенно *при удалении* имен каталогов. Предположим, что один процесс пытается найти данные файла по его полному символическому имени, последовательно проходя компонент за компонентом, а другой процесс удаляет каталог, имя которого входит в путь поиска. Допустим, процесс A делает разбор имени "a/b/c/d" и приостанавливается во время получе-

ния индексного узла для файла "c". Он может приостановиться при попытке заблокировать индексный узел или при попытке обратиться к дисковому блоку, где этот индексный узел хранится. Если процессу B нужно удалить связь для каталога с именем "c", он может приостановиться по той же самой причине, что и процесс A. Пусть ядро впоследствии решит возобновить процесс B раньше процесса A. Прежде чем процесс A продолжит свое выполнение, процесс B завершится, удалив связь каталога "c" и его содержимое по этой связи. Позднее процесс A попытается обратиться к несуществующему индексному узлу, который уже был удален. Алгоритм поиска файла, проверяющий в первую очередь неравенство значения счетчика связей нулю, должен сообщить об ошибке.

Можно привести и другие примеры, которые демонстрируют необходимость тщательного проектирования файловой системы для ее последующей надежной работы.

Надежность файловой системы

Жизнь полна неприятных неожиданностей, а разрушение файловой системы зачастую более опасно, чем разрушение компьютера. Поэтому файловые системы должны разрабатываться с учетом подобной возможности. Помимо очевидных решений (например своевременное дублирование информации (backup)) файловые системы современных ОС содержат специальные средства для поддержки собственной совместимости.

Целостность файловой системы

Важный аспект надежной работы файловой системы – контроль ее целостности. В результате файловых операций блоки диска могут считываться в память, модифицироваться и затем записываться на диск. Причем многие файловые операции затрагивают сразу несколько объектов файловой системы. Например, копирование файла предполагает выделение ему блоков диска, формирование индексного узла, изменение содержимого каталога и т. д. В течение короткого периода времени между этими шагами информация в файловой системе оказывается несогласованной.

И если вследствие непредсказуемого останова системы на диске будут сохранены изменения только для части этих объектов (нарушена атомарность файловой операции), файловая система на диске может быть оставлена в несовместимом состоянии. В результате могут возникнуть нарушения логики работы с данными, например появиться «потерянные» блоки диска, которые не принадлежат ни одному файлу и в то же время

помечены как занятые, или, наоборот, блоки, помеченные как свободные, но в то же время занятые (на них есть ссылка в индексном узле) или другие нарушения.

В современных ОС предусмотрены меры, которые позволяют свести к минимуму ущерб от порчи файловой системы и затем полностью или частично восстановить ее целостность.

Порядок выполнения операций

Очевидно, что для правильного функционирования файловой системы значимость отдельных данных неравноценна. Искажение содержимого пользовательских файлов не приводит к серьезным (с точки зрения целостности файловой системы) последствиям, тогда как несоответствия в файлах, содержащих управляющую информацию (директории, индексные узлы, суперблок и т. п.), могут быть катастрофическими. Поэтому должен быть тщательно продуман порядок выполнения операций со структурами данных файловой системы.

Рассмотрим пример создания жесткой связи для файла [Робачевский, 1999]. Для этого файловой системе необходимо выполнить следующие операции:

- создать новую запись в каталоге, указывающую на индексный узел файла;
- увеличить счетчик связей в индексном узле.

Если аварийный останов произошел между 1-й и 2-й операциями, то в каталогах файловой системы будут существовать два имени файла, адресующих индексный узел со значением счетчика связей, равному 1. Если теперь будет удалено одно из имен, это приведет к удалению файла как такового. Если же порядок операций изменен и, как прежде, останов произошел между первой и второй операциями, файл будет иметь несуществующую жесткую связь, но существующая запись в каталоге будет правильной. Хотя это тоже является ошибкой, но ее последствия менее серьезны, чем в предыдущем случае.

Журнализация

Другим средством поддержки целостности является заимствованный из систем управления базами данных прием, называемый *журнализация* (*иногда употребляется термин «журналирование»*). Последовательность действий с объектами во время файловой операции протоколируется, и если произошел останов системы, то, имея в наличии протокол, можно осуществить откат системы назад в исходное целостное состояние, в котором она пребывала до начала операции. Подобная избыточность может

стоить дорого, но она оправданна, так как в случае отказа позволяет реконструировать потерянные данные.

Для отката необходимо, чтобы для каждой протоколируемой в журнале операции существовала обратная. Например, для каталогов и реляционных СУБД это именно так. По этой причине, в отличие от СУБД, в файловых системах протоколируются не все изменения, а лишь изменения метаданных (индексных узлов, записей в каталогах и др.). Изменения в данных пользователя в протокол не заносятся. Кроме того, если протоколировать изменения пользовательских данных, то этим будет нанесен серьезный ущерб производительности системы, поскольку кэширование потеряет смысл.

Журнализация реализована в NTFS, Ext3FS, ReiserFS и других системах. Чтобы подчеркнуть сложность задачи, нужно отметить, что существуют не вполне очевидные проблемы, связанные с процедурой отката. Например, отмена одних изменений может затрагивать данные, уже использованные другими файловыми операциями. Это означает, что такие операции также должны быть отменены. Данная проблема получила название каскадного отката транзакций [Брукшир, 2001].

Проверка целостности файловой системы при помощи утилит

Если же нарушение все же произошло, то для устранения проблемы несовместимости можно прибегнуть к утилитам (fsck, chkdsk, scandisk и др.), которые проверяют целостность файловой системы. Они могут запускаться после загрузки или после сбоя и осуществляют многократное сканирование разнообразных структур данных файловой системы в поисках противоречий.

Возможны также эвристические проверки. Например, нахождение индексного узла, номер которого превышает их число на диске или поиск в пользовательских директориях файлов, принадлежащих суперпользователю.

К сожалению, приходится констатировать, что *не существует никаких средств, гарантирующих абсолютную сохранность* информации в файлах, и в тех ситуациях, когда целостность информации нужно гарантировать с высокой степенью надежности, прибегают к дорогостоящим процедурам дублирования.

Управление «плохими» блоками

Наличие дефектных блоков на диске – обычное дело. Внутри блока наряду с данными хранится контрольная сумма данных. Под «плохими» блоками обычно понимают блоки диска, для которых вычисленная конт-

рольная сумма считываемых данных не совпадает с хранимой контрольной суммой. Дефектные блоки обычно появляются в процессе эксплуатации. Иногда они уже имеются при поставке вместе со списком, так как очень затруднительно для поставщиков сделать диск полностью свободным от дефектов. Рассмотрим два решения проблемы дефектных блоков – одно на уровне аппаратуры, другое на уровне ядра ОС.

Первый способ – хранить список плохих блоков в контроллере диска. Когда контроллер инициализируется, он читает плохие блоки и замещает дефектный блок резервным, помечая отображение в списке плохих блоков. Все реальные запросы будут идти к резервному блоку. Следует иметь в виду, что при этом механизм подъемника (наиболее распространенный механизм обработки запросов к блокам диска) будет работать неэффективно. Дело в том, что существует стратегия очередности обработки запросов к диску (подробнее см. лекцию «ввод-вывод»). Стратегия диктует направление движения считывающей головки диска к нужному цилиндру. Обычно резервные блоки размещаются на внешних цилиндрах. Если плохой блок расположен на внутреннем цилиндре и контроллер осуществляет подстановку прозрачным образом, то кажущееся движение головки будет осуществляться к внутреннему цилиндру, а фактическое – к внешнему. Это является нарушением стратегии и, следовательно, минусом данной схемы.

Решение на уровне ОС может быть следующим. Прежде всего, необходимо тщательно сконструировать файл, содержащий дефектные блоки. Тогда они изымаются из списка свободных блоков. Затем нужно каким-то образом скрыть этот файл от прикладных программ.

Производительность файловой системы

Поскольку обращение к диску – операция относительно медленная, минимизация количества таких обращений является ключевой задачей всех алгоритмов, работающих с внешней памятью. Наиболее типичная техника повышения скорости работы с диском – *кэширование*.

Кэширование

Кэш диска представляет собой буфер в оперативной памяти, содержащий ряд блоков диска (см. рис. 12.12). Если имеется запрос на чтение/запись блока диска, то сначала производится проверка на предмет наличия этого блока в кэше. Если блок в кэше имеется, то запрос удовлетворяется из кэша, в противном случае запрошенный блок считывается в кэш с диска. Сокращение количества дисковых операций оказывается возможным вследствие присущего ОС свойства локальности (о свойстве локаль-

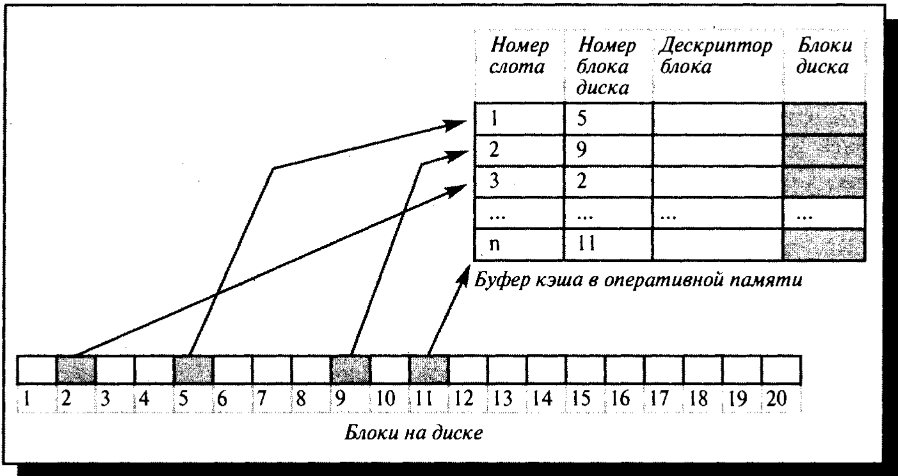


Рис. 12.12. Структура блочного кэша

ности много говорилось в лекциях, посвященных описанию работы системы управления памятью).

Аккуратная реализация кэширования требует решения нескольких проблем.

Во-первых, емкость буфера кэша ограничена. Когда блок должен быть загружен в заполненный буфер кэша, возникает *проблема замещения* блоков, то есть отдельные блоки должны быть удалены из него. Здесь работают те же стратегии и те же FIFO, Second Chance и LRU-алгоритмы замещения, что и при выталкивании страниц памяти.

Замещение блоков должно осуществляться с учетом их важности для файловой системы. Блоки должны быть разделены *на категории*, например: блоки индексных узлов, блоки косвенной адресации, блоки директорий, заполненные блоки данных и т. д., и в зависимости от принадлежности блока к той или иной категории можно применять к ним разную стратегию замещения.

Во-вторых, поскольку кэширование использует механизм отложенной записи, при котором модификация буфера не вызывает немедленной записи на диск, серьезной проблемой является «старение» информации в дисковых блоках, образы которых находятся в буферном кэше. Несвоевременная *синхронизация* буфера кэша и диска может привести к очень нежелательным последствиям в случае отказов оборудования или программного обеспечения. Поэтому стратегия и порядок отображения информации из кэша на диск должна быть тщательно продумана.

Так, блоки, существенные для совместимости файловой системы (блоки индексных узлов, блоки косвенной адресации, блоки директо-

рий), должны быть переписаны на диск немедленно, независимо от того, в какой части LRU-цепочки они находятся. Необходимо тщательно выбрать порядок такого переписывания.

В Unix имеется для этого вызов SYNC, который заставляет все модифицированные блоки записываться на диск немедленно. Для синхронизации содержимого кэша и диска периодически запускается фоновый процесс-демон. Кроме того, можно организовать синхронный режим работы с отдельными файлами, задаваемый при открытии файла, когда все изменения в файле немедленно сохраняются на диске.

Наконец, проблема конкуренции процессов на доступ к блокам кэша решается ведением списков блоков, пребывающих в различных состояниях, и отметкой о состоянии блока в его дескрипторе. Например, блок может быть заблокирован, участвовать в операции ввода-вывода, а также иметь список процессов, ожидающих освобождения данного блока.

Оптимальное размещение информации на диске

Кэширование – не единственный способ увеличения производительности системы. Другая важная техника – сокращение количества движений считывающей головки диска за счет разумной стратегии размещения информации. Например, массив индексных узлов в Unix стараются разместить на средних дорожках. Также имеет смысл размещать индексные узлы поблизости от блоков данных, на которые они ссылаются и т.д.

Кроме того, рекомендуется периодически осуществлять дефрагментацию диска (сборку мусора), поскольку в популярных методиках выделения дисковых блоков (за исключением, может быть, FAT) принцип локальности не работает, и последовательная обработка файла требует обращения к различным участкам диска.

Реализация некоторых операций над файлами

В предыдущей лекции перечислены основные операции над файлами. В данном разделе будет описан порядок работы некоторых системных вызовов для работы с файловой системой, следуя главным образом [Bach, 1986], с учетом совокупности введенных в данной лекции понятий.

Системные вызовы, работающие с символическим именем файла

Системные вызовы, связывающие pathname с дескриптором файла

Это функции создания и открытия файла. Например, в ОС Unix

```
fd = creat(pathname,modes);
fd = open(pathname,flags,modes);
```

Другие операции над файлами, такие как чтение, запись, позиционирование головок чтения-записи, воспроизведение дескриптора файла, установка параметров ввода-вывода, определение статуса файла и закрытие файла, используют значение полученного дескриптора файла.

Рассмотрим работу системного вызова open.

Логическая файловая подсистема просматривает файловую систему в поисках файла по его имени. Она проверяет права на открытие файла и выделяет открываемому файлу запись в таблице файлов. Запись таблицы файлов содержит указатель на индексный узел открытого файла. Ядро выделяет запись в личной (закрытой) таблице в адресном пространстве процесса, выделенном процессу (таблица эта называется таблицей пользовательских дескрипторов открытых файлов) и запоминает указатель на данную запись. В роли указателя выступает дескриптор файла, возвращаемый пользователю. Запись в таблице пользовательских файлов указывает на запись в глобальной таблице файлов (см. рис. 12.13).

Первые три пользовательских дескриптора (0, 1 и 2) именуются дескрипторами файлов стандартного ввода, стандартного вывода и стандартного файла ошибок. Процессы в системе Unix по договоренности используют дескриптор файла стандартного ввода при чтении вводимой информации, дескриптор файла стандартного вывода при записи выводимой информации и дескриптор стандартного файла ошибок для записи сообщений об ошибках.

Связывание файла

Системная функция `link` связывает файл с новым именем в структуре каталогов файловой системы, создавая для существующего индекса новую запись в каталоге. Синтаксис вызова функции `link`

```
link(source file name, target file name);
```

где `source file name` – существующее имя файла, а `target file name` – новое (дополнительное) имя, присваиваемое файлу после выполнения функции `link`.

Сначала ОС определяет местонахождение индекса исходного файла и увеличивает значение счетчика связей в индексном узле. Затем ядро ищет файл с новым именем; если он существует, функция `link` завершается неудачно, и ядро восстанавливает прежнее значение счетчика связей, измененное ранее. В противном случае ядро находит в родительском ка-

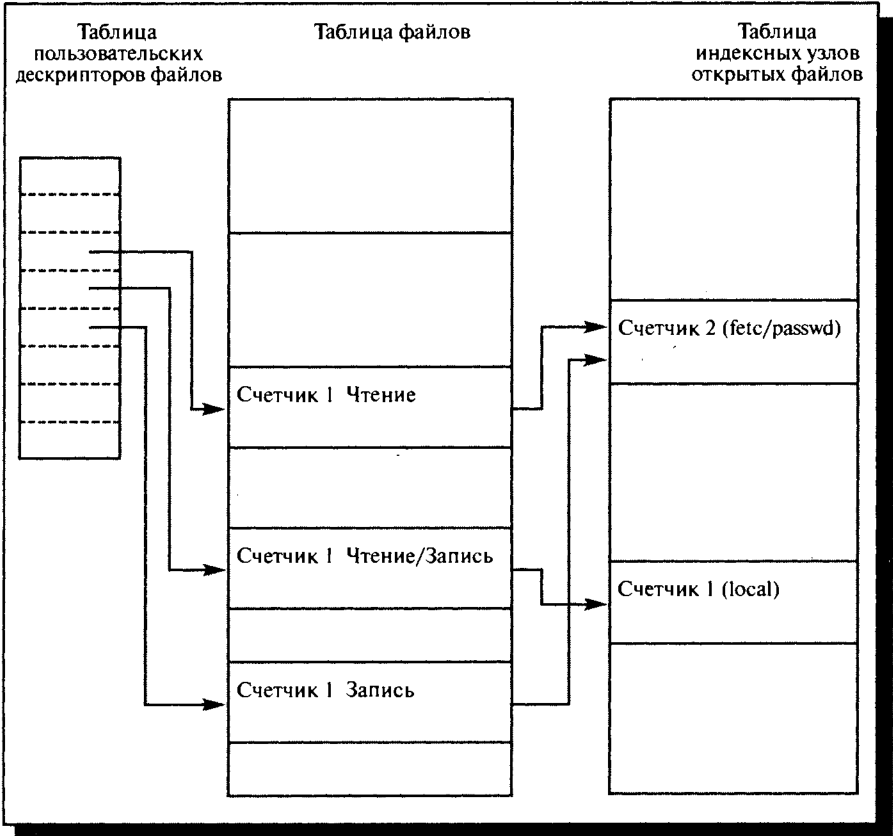


Рис. 12.13. Структуры данных после открытия файлов

талоге свободную запись для файла с новым именем, записывает в нее новое имя и номер индекса исходного файла.

Удаление файла

В Unix системная функция `unlink` удаляет из каталога точку входа для файла. Синтаксис вызова функции `unlink`

```
unlink(pathname);
```

Если удаляемое имя является последней связью файла с каким-либо каталогом, ядро в итоге освобождает все информационные блоки файла.

Однако если у файла было несколько связей, он остается все еще доступным под другими именами.

Для того чтобы забрать дисковые блоки, ядро в цикле просматривает таблицу содержимого индексного узла, освобождая все блоки прямой адресации немедленно. Что касается блоков косвенной адресации, то ядро освобождает все блоки, появляющиеся на различных уровнях косвенности, рекурсивно, причем в первую очередь освобождаются блоки с меньшим уровнем.

Системные вызовы, работающие с файловым дескриптором

Открытый файл может использоваться для чтения и записи последовательностей байтов. Для этого поддерживаются два системных вызова `read` и `write`, работающие с файловым дескриптором (или handle в терминологии Microsoft), полученным при ранее выполненных системных вызовах `open` или `creat`.

Функции ввода-вывода из файла

Системный вызов `read` выполняет чтение обычного файла

```
number = read(fd,buffer,count);
```

где `fd` – дескриптор файла, возвращаемый функцией open, `buffer` – адрес структуры данных в пользовательском процессе, где будут размещаться считанные данные в случае успешного завершения выполнения функции `read`, `count` – количество байтов, которые пользователю нужно прочитать, `number` – количество фактически прочитанных байтов.

Синтаксис вызова системной функции `write` (писать)

```
number = write(fd,buffer,count);
```

где переменные `fd`, `buffer`, `count` и `number` имеют тот же смысл, что и для вызова системной функции `read`. Алгоритм записи в обычный файл похож на алгоритм чтения из обычного файла. Однако если в файле отсутствует блок, соответствующий смещению в байтах до места, куда должна производиться запись, ядро выделяет блок и присваивает ему номер в соответствии с точным указанием места в таблице содержимого индексного узла.

Обычное использование системных функций `read` и `write` обеспечивает последовательный доступ к файлу, однако процессы могут использовать вызов системной функции `lseek` для указания места в файле, где будет производиться ввод-вывод, и осуществления произвольного доступа к файлу. Синтаксис вызова системной функции

```
position = lseek(fd,offset,reference);
```

где `fd` – дескриптор файла, идентифицирующий файл, `offset` – смещение в байтах, а `reference` указывает, является ли значение `offset` смещением от начала файла, смещением от текущей позиции ввода-вывода или смещением от конца файла. Возвращаемое значение, `position`, является смещением в байтах до места, где будет начинаться следующая операция чтения или записи.

Современные архитектуры файловых систем

Современные ОС предоставляют пользователю возможность работать сразу с несколькими файловыми системами (Linux работает с Ext2fs, FAT и др.). Файловая система в традиционном понимании становится частью более общей многоуровневой структуры (см. рис. 12.14).

На верхнем уровне располагается так называемый диспетчер файловых систем (например, в Windows 95 этот компонент называется installable filesystem manager). Он связывает запросы прикладной программы с конкретной файловой системой.

Каждая файловая система (иногда говорят – драйвер файловой системы) на этапе инициализации регистрируется у диспетчера, сообщая ему точки входа для последующих обращений к данной файловой системе.

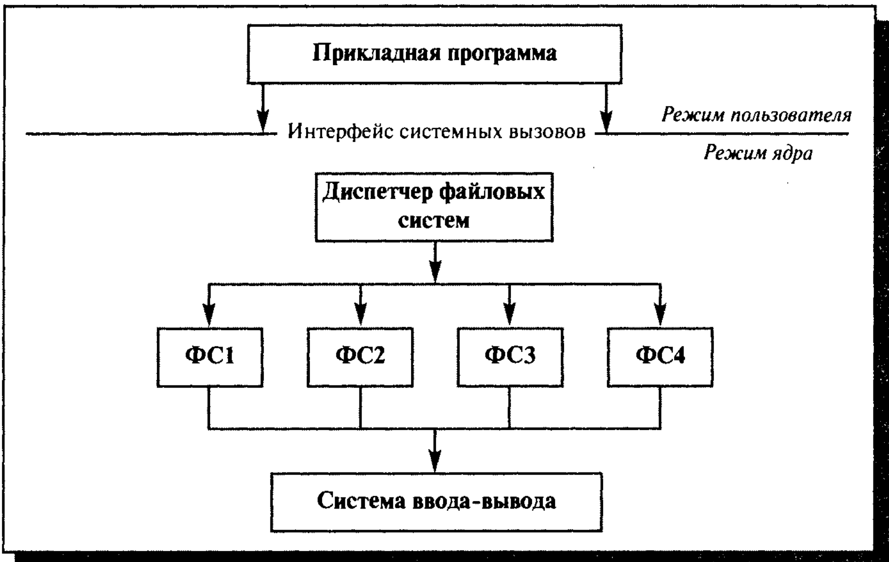


Рис. 12.14. Архитектура современной файловой системы

Та же идея поддержки нескольких файловых систем в рамках одной ОС может быть реализована по-другому, например исходя из концепции виртуальной файловой системы. Виртуальная файловая система (vfs) представляет собой независимый от реализации уровень и опирается на реальные файловые системы (s5fs, ufs, FAT, NFS, FFS. Ext2fs ...). При этом возникают структуры данных виртуальной файловой системы типа виртуальных индексных узлов vnode, которые обобщают индексные узлы конкретных систем.

Заключение

Реализация файловой системы связана с такими вопросами, как поддержка понятия логического блока диска, связывания имени файла и блоков его данных, проблемами разделения файлов и проблемами управления дискового пространства.

Наиболее распространенные способы выделения дискового пространства: непрерывное выделение, организация связного списка и система с индексными узлами.

Файловая система часто реализуется в виде слоеной модульной структуры. Нижние слои имеют дело с оборудованием, а верхние – с символическими именами и логическими свойствами файлов.

Директории могут быть организованы различными способами и могут хранить атрибуты файла и адреса блоков файлов, а иногда для этого предназначается специальная структура (индексные узлы).

Проблемы надежности и производительности файловой системы – важнейшие аспекты ее дизайна.

Часть V. Ввод-вывод

Лекция 13. Система управления вводом-выводом

В лекции рассматриваются основные физические и логические принципы организации ввода-вывода в вычислительных системах.

Ключевые слова: локальная магистраль, шина адреса, шина данных, шина управления, порт ввода-вывода, адресное пространство ввода-вывода, контроллер устройства, polling, прерывание, контроллер прерывания, регистр состояния, регистр управления, регистры входных и выходных данных, исключительная ситуация, программное прерывание, прямой доступ к памяти (DMA), символьное устройство, блочное устройство, базовая подсистема ввода-вывода, драйвер ввода-вывода, асинхронный системный вызов, блокирующийся системный вызов, неблокирующийся системный вызов, буферизация, кэширование, алгоритмы планирования запросов к жесткому диску – FCFS, SSTF, SCAN, LOOK, C-SCAN, C-LOOK.

Функционирование любой вычислительной системы обычно сводится к выполнению двух видов работы: обработке информации и операций по осуществлению ее ввода-вывода. Поскольку в рамках модели, принятой в данном курсе, все, что выполняется в вычислительной системе, организовано как набор процессов, эти два вида работы выполняются процессами. Процессы занимаются обработкой информации и выполнением операций ввода-вывода.

Содержание понятий «обработка информации» и «операции ввода-вывода» зависит от того, с какой точки зрения мы смотрим на них. С точки зрения программиста, под «обработкой информации» понимается выполнение команд процессора над данными, лежащими в памяти независимо от уровня иерархии – в регистрах, кэше, оперативной или вторичной памяти. Под «операциями ввода-вывода» программист понимает обмен данными между памятью и устройствами, внешними по отношению к памяти и процессору, такими как магнитные ленты, диски, монитор, клавиатура, таймер. С точки зрения операционной системы «обработкой информации» являются только операции, совершаемые процессором над данными, находящимися в памяти на уровне иерархии не ниже, чем оперативная память. Все остальное относится к «операциям ввода-вывода». Чтобы выполнять операции над данными, временно расположенными во вторичной памяти, операционная система,

как мы обсуждали в части III нашего курса, сначала производит их подкачку в оперативную память, и лишь затем процессор совершает необходимые действия.

Объяснение того, что именно делает процессор при обработке информации, как он решает задачу и какой алгоритм выполняет, не входит в задачи нашего курса. Это скорее относится к курсу «Алгоритмы и структуры данных», с которого обычно начинается изучение информатики. Как операционная система управляет обработкой информации, мы разобрали в части II, в деталях описав два состояния процессов – ***исполнение*** (а что его описывать-то?) и ***готовность*** (очереди планирования и т. д.), а также правила, по которым осуществляется перевод процессов из одного состояния в другое (алгоритмы планирования процессов).

Данная лекция будет посвящена второму виду работы вычислительной системы – операциям ввода-вывода. Мы разберем, что происходит в компьютере при выполнении операций ввода-вывода, и как операционная система управляет их выполнением. При этом для простоты будем считать, что объем оперативной памяти в вычислительной системе достаточно большой, т. е. все процессы полностью располагаются в оперативной памяти, и поэтому понятие «операция ввода-вывода» с точки зрения операционной системы и с точки зрения пользователя означает одно и то же. Такое предположение не снижает общности нашего рассмотрения, так как подкачка информации из вторичной памяти в оперативную память и обратно обычно строится по тому же принципу, что и все операции ввода-вывода.

Прежде чем говорить о работе операционной системы при осуществлении операций ввода-вывода, нам придется вспомнить некоторые сведения из курса «Архитектура современных ЭВМ и язык Ассемблера», чтобы понять, как осуществляется передача информации между оперативной памятью и внешним устройством и почему для подключения к вычислительной системе новых устройств ее не требуется перепроектировать.

Физические принципы организации ввода-вывода

Существует много разнообразных устройств, которые могут взаимодействовать с процессором и памятью: таймер, жесткие диски, клавиатура, дисплеи, мышь, модемы и т. д., вплоть до устройств отображения и ввода информации в авиационно-космических тренажерах. Часть этих устройств может быть встроена внутрь корпуса компьютера, часть – вынесена за его пределы и общаться с компьютером через различные линии связи: кабельные, оптоволоконные, радиорелейные, спутниковые и т. д. Конкретный набор устройств и способы их подключения определяются целями функционирования вычислительной системы, желаниями и фи-

нансовыми возможностями пользователя. Несмотря на все многообразие устройств, управление их работой и обмен информацией с ними строятся на относительно небольшом наборе принципов, которые мы постараемся разобрать в этом разделе.

Общие сведения об архитектуре компьютера

В простейшем случае процессор, память и многочисленные внешние устройства связаны большим количеством электрических соединений – *линий*, которые в совокупности принято называть *локальной магистралью* компьютера. Внутри локальной магистрали линии, служащие для передачи сходных сигналов и предназначенные для выполнения сходных функций, принято группировать в *шины*. При этом понятие шины включает в себя не только набор проводников, но и набор жестко заданных протоколов, определяющий перечень сообщений, который может быть передан с помощью электрических сигналов по этим проводникам. В современных компьютерах выделяют как минимум три шины:

- шину данных, состоящую из линий данных и служащую для передачи информации между процессором и памятью, процессором и устройствами ввода-вывода, памятью и внешними устройствами;
- адресную шину, состоящую из линий адреса и служащую для задания адреса ячейки памяти или указания устройства ввода-вывода, участвующих в обмене информацией;
- шину управления, состоящую из линий управления локальной магистралью и линий ее состояния, определяющих поведение локальной магистрали. В некоторых архитектурных решениях линии состояния выносятся из этой шины в отдельную шину состояния.

Количество линий, входящих в состав шины, принято называть *разрядностью* (*шириной*) этой шины. Ширина адресной шины, например, определяет максимальный размер оперативной памяти, которая может быть установлена в вычислительной системе. Ширина шины данных определяет максимальный объем информации, которая за один раз может быть получена или передана по этой шине.

Операции обмена информацией осуществляются при одновременном участии всех шин. Рассмотрим, к примеру, действия, которые должны быть выполнены для передачи информации из процессора в память. В простейшем случае необходимо выполнить три действия:

1. На адресной шине процессор должен выставить сигналы, соответствующие адресу ячейки памяти, в которую будет осуществляться передача информации.
2. На шину данных процессор должен выставить сигналы, соответствующие информации, которая должна быть записана в память.

3. После выполнения действий 1 и 2 на шину управления выставляются сигналы, соответствующие операции записи и работе с памятью, что приведет к занесению необходимой информации по нужному адресу.

Естественно, что приведенные выше действия являются необходимыми, но недостаточными при рассмотрении работы конкретных процессоров и микросхем памяти. Конкретные архитектурные решения могут требовать дополнительных действий: например, выставления на шину управления сигналов частичного использования шины данных (для передачи меньшего количества информации, чем позволяет ширина этой шины); выставления сигнала готовности магистрали после завершения записи в память, разрешающего приступить к новой операции, и т. д. Однако общие принципы выполнения операции записи в память остаются неизменными.

В то время как память легко можно представить себе в виде последовательности пронумерованных адресами ячеек, локализованных внутри одной микросхемы или набора микросхем, к устройствам ввода-вывода подобный подход неприменим. Внешние устройства разнесены пространственно и могут подключаться к локальной магистрали в одной точке или множестве точек, получивших название *портов ввода-вывода*. Тем не менее, точно так же, как ячейки памяти взаимно однозначно отображались в адресное пространство памяти, порты ввода-вывода можно взаимно однозначно отобразить в другое адресное пространство — адресное пространство ввода-вывода. При этом каждый порт ввода-вывода получает свой номер или адрес в этом пространстве. В некоторых случаях, когда адресное пространство памяти (размер которого определяется шириной адресной шины) задействовано не полностью (остались адреса, которым не соответствуют физические ячейки памяти) и протоколы работы с внешним устройством совместимы с протоколами работы с памятью, часть портов ввода-вывода может быть отображена непосредственно в адресное пространство памяти (так, например, поступают с видеопамятью дисплеев). Правда, тогда эти порты уже не принято называть портами. Надо отметить, что при отображении портов в адресное пространство памяти для организации доступа к ним в полной мере могут быть задействованы существующие механизмы защиты памяти без организации специальных защитных устройств.

В ситуации прямого отображения портов ввода-вывода в адресное пространство памяти действия, необходимые для записи информации и управляющих команд в эти порты или для чтения данных из них и их состояний, ничем не отличаются от действий, производимых для передачи информации между оперативной памятью и процессором, и для их выполнения применяются те же самые команды. Если же порт отображен в адресное пространство ввода-вывода, то процесс обмена информацией

инициируется специальными командами ввода-вывода и включает в себя несколько другие действия. Например, для передачи данных в порт необходимо выполнить следующее:

- На адресной шине процессор должен выставить сигналы, соответствующие адресу порта, в который будет осуществляться передача информации, в адресном пространстве ввода-вывода.
- На шину данных процессор должен выставить сигналы, соответствующие информации, которая должна быть передана в порт.
- После выполнения действий 1 и 2 на шину управления выставляются сигналы, соответствующие операции записи и работе с устройствами ввода-вывода (переключение адресных пространств!), что приведет к передаче необходимой информации в нужный порт.

Существенное отличие памяти от устройств ввода-вывода заключается в том, что занесение информации в память является окончанием операции записи, в то время как занесение информации в порт зачастую представляет собой инициализацию реального совершения операции ввода-вывода. Что именно должны делать устройства, приняв информацию через свой порт, и каким именно образом они должны поставлять информацию для чтения из порта, определяется электронными схемами устройств, получившими название *контроллеров*. Контроллер может непосредственно управлять отдельным устройством (например, контроллер диска), а может управлять несколькими устройствами, связываясь с их контроллерами посредством специальных шин ввода-вывода (шина IDE, шина SCSI и т. д.).

Современные вычислительные системы могут иметь разнообразную архитектуру, множество шин и магистралей, мосты для перехода информации от одной шины к другой и т. п. Для нас сейчас важными являются только следующие моменты:

- Устройства ввода-вывода подключаются к системе через порты.
- Могут существовать два адресных пространства: пространство памяти и пространство ввода-вывода.
- Порты, как правило, отображаются в адресное пространство ввода-вывода и иногда – непосредственно в адресное пространство памяти.
- Использование того или иного адресного пространства определяется типом команды, выполняемой процессором, или типом ее операндов.
- Физическим управлением устройством ввода-вывода, передачей информации через порт и выставлением некоторых сигналов на магистрали занимается контроллер устройства.

Именно единообразие подключения внешних устройств к вычислительной системе является одной из составляющих идеологии, позволяющих добавлять новые устройства без перепроектирования всей системы.

Структура контроллера устройства

Контроллеры устройств ввода-вывода весьма различны как по своему внутреннему строению, так и по исполнению (от одной микросхемы до специализированной вычислительной системы со своим процессором, памятью и т. д.), поскольку им приходится управлять совершенно разными приборами. Не вдаваясь в детали этих различий, мы выделим некоторые общие черты контроллеров, необходимые им для взаимодействия с вычислительной системой. Обычно каждый контроллер имеет по крайней мере четыре внутренних регистра, называемых регистрами ***состояния***, ***управления***, ***входных данных*** и ***выходных данных***. Для доступа к содержимому этих регистров вычислительная система может использовать один или несколько портов, что для нас не существенно. Для простоты изложения будем считать, что каждому регистру соответствует свой порт.

Регистр ***состояния*** содержит биты, значение которых определяется состоянием устройства ввода-вывода и которые доступны только для чтения вычислительной системой. Эти биты индицируют завершение выполнения текущей команды на устройстве (***бит занятости***), наличие очередного данного в регистре выходных данных (***бит готовности данных***), возникновение ошибки при выполнении команды (***бит ошибки***) и т. д.

Регистр ***управления*** получает данные, которые записываются вычислительной системой для инициализации устройства ввода-вывода или выполнения очередной команды, а также изменения режима работы устройства. Часть битов в этом регистре может быть отведена под код выполняемой команды, часть битов будет кодировать режим работы устройства, бит ***готовности команды*** свидетельствует о том, что можно приступить к ее выполнению.

Регистр ***выходных данных*** служит для помещения в него данных для чтения вычислительной системой, а регистр ***входных данных*** предназначен для помещения в него информации, которая должна быть выведена на устройство. Обычно емкость этих регистров не превышает ширину линии данных (а чаще всего меньше ее), хотя некоторые контроллеры могут использовать в качестве регистров очередь FIFO для буферизации поступающей информации.

Разумеется, набор регистров и составляющих их битов приблизителен, он призван послужить нам моделью для описания процесса передачи информации от вычислительной системы к внешнему устройству и обратно, но в том или ином виде он обычно присутствует во всех контроллерах устройств.

Опрос устройств и прерывания. Исключительные ситуации и системные вызовы

Построив модель контроллера и представляя себе, что скрывается за словами «прочитать информацию из порта» и «записать информацию в порт», мы готовы к рассмотрению процесса взаимодействия устройства и процессора. Как и в предыдущих случаях, примером нам послужит команда записи, теперь уже записи или вывода данных на внешнее устройство. В нашей модели для вывода информации, помещающейся в регистр ***входных данных***, без проверки успешности вывода процессор и контроллер должны связываться следующим образом:

1) Процессор в цикле читает информацию из порта регистра ***состояний*** и проверяет значение ***бита занятости***. Если ***бит занятости*** установлен, то это означает, что устройство еще не завершило предыдущую операцию, и процессор уходит на новую итерацию цикла. Если ***бит занятости*** сброшен, то устройство готово к выполнению новой операции, и процессор переходит на следующий шаг.
2) Процессор записывает код команды вывода в порт регистра ***управления***.
3) Процессор записывает данные в порт регистра ***входных данных***.
4) Процессор устанавливает ***бит готовности команды***. В следующих шагах процессор не задействован.
5) Когда контроллер замечает, что ***бит готовности команды*** установлен, он устанавливает ***бит занятости***.
6) Контроллер анализирует код команды в регистре ***управления*** и обнаруживает, что это команда вывода. Он берет данные из регистра ***входных данных*** и инициирует выполнение команды.
7) После завершения операции контроллер обнуляет ***бит готовности команды***.
8) При успешном завершении операции контроллер обнуляет ***бит ошибки*** в регистре состояния, при неудачном завершении команды – устанавливает его.
9) Контроллер сбрасывает ***бит занятости***.

При необходимости вывода новой порции информации все эти шаги повторяются. Если процессор интересует, корректно или некорректно была выведена информация, то после шага 4 он должен в цикле считывать информацию из порта регистра ***состояний*** до тех пор, пока не будет сброшен ***бит занятости*** устройства, после чего проанализировать состояние ***бита ошибки***.

Как видим, на первом шаге (и, возможно, после шага 4) процессор ожидает освобождения устройства, непрерывно опрашивая значение ***бита занятости***. Такой способ взаимодействия процессора и контроллера получил название *polling* или, в русском переводе, ***способа опроса устройств***.

Если скорости работы процессора и устройства ввода-вывода примерно равны, то это не приводит к существенному уменьшению полезной работы, совершаемой процессором. Если же скорость работы устройства существенно меньше скорости процессора, то указанная техника резко снижает производительность системы и необходимо применять другой архитектурный подход.

Для того чтобы процессор не дожидался состояния готовности устройства ввода-вывода в цикле, а мог выполнять в это время другую работу, необходимо, чтобы устройство само умело сигнализировать процессору о своей готовности. Технический механизм, который позволяет внешним устройствам оповещать процессор о завершении команды вывода или команды ввода, получил название *механизма прерываний*.

В простейшем случае для реализации механизма прерываний необходимо к имеющимся у нас шинам локальной магистрали добавить еще одну линию, соединяющую процессор и устройства ввода-вывода – линию прерываний. По завершении выполнения операции внешнее устройство выставляет на эту линию специальный сигнал, по которому процессор после выполнения очередной команды (или после завершения очередной итерации при выполнении цепочечных команд, т. е. команд, повторяющихся циклически со сдвигом по памяти) изменяет свое поведение. Вместо выполнения очередной команды из потока команд он частично сохраняет содержимое своих регистров и переходит на выполнение программы обработки прерывания, расположенной по заранее оговоренному адресу. При наличии только одной линии прерываний процессор при выполнении этой программы должен опросить состояние всех устройств ввода-вывода, чтобы определить, от какого именно устройства пришло прерывание (polling прерываний!), выполнить необходимые действия (например, вывести в это устройство очередную порцию информации или перевести соответствующий процесс из состояния ***ожидание*** в состояние ***готовность***) и сообщить устройству, что прерывание обработано (снять прерывание).

В большинстве современных компьютеров процессор стараются полностью освободить от необходимости опроса внешних устройств, в том числе и от определения с помощью опроса устройства, сгенерировавшего сигнал прерывания. Устройства сообщают о своей готовности процессору не напрямую, а через специальный контроллер прерываний, при этом для общения с процессором он может использовать не одну линию, а целую шину прерываний. Каждому устройству присваивается свой номер прерывания, который при возникновении прерывания контроллер прерывания заносит в свой регистр состояния и, возможно, после распознавания процессором сигнала прерывания и получения от него специального запроса выставляет на шину прерываний или шину данных для чтения процессором. Номер прерывания обычно служит индек-

сом в специальной таблице прерываний, хранящейся по адресу, задаваемому при инициализации вычислительной системы, и содержащей адреса программ обработки прерываний – *векторы* прерываний. Для распределения устройств по номерам прерываний необходимо, чтобы от каждого устройства к контроллеру прерываний шла специальная линия, соответствующая одному номеру прерывания. При наличии множества устройств такое подключение становится невозможным, и на один проводник (один номер прерывания) подключается несколько устройств. В этом случае процессор при обработке прерывания все равно вынужден заниматься опросом устройств для определения устройства, выдавшего прерывание, но в существенно меньшем объеме. Обычно при установке в систему нового устройства ввода-вывода требуется аппаратно или программно определить, каким будет номер прерывания, вырабатываемый этим устройством.

Рассматривая кооперацию процессов и взаимоисключения, мы говорили о существовании критических секций внутри ядра операционной системы, при выполнении которых необходимо исключить всякие прерывания от внешних устройств. Для запрещения прерываний, а точнее, для невосприимчивости процессора к внешним прерываниям обычно существуют специальные команды, которые могут маскировать (запрещать) все или некоторые из прерываний устройств ввода-вывода. В то же время определенные кризисные ситуации в вычислительной системе (например, неустранимый сбой в работе оперативной памяти) должны требовать ее немедленной реакции. Такие ситуации вызывают прерывания, которые невозможно замаскировать или запретить и которые поступают в процессор по специальной линии шины прерываний, называемой линией немаскируемых прерываний (NMI – Non-Maskable Interrupt).

Не все внешние устройства являются одинаково важными с точки зрения вычислительной системы («все животные равны, но некоторые равнее других»). Соответственно, некоторые прерывания являются более существенными, чем другие. Контроллер прерываний обычно позволяет устанавливать приоритеты для прерываний от внешних устройств. При почти одновременном возникновении прерываний от нескольких устройств (во время выполнения одной и той же команды процессора) процессору сообщается номер наиболее приоритетного прерывания для его обслуживания в первую очередь. Менее приоритетное прерывание при этом не пропадает, о нем процессору будет доложено после обработки более приоритетного прерывания. Более того, при обработке возникшего прерывания процессор может получить сообщение о возникновении прерывания с более высоким приоритетом и переключиться на его обработку.

Механизм обработки прерываний, по которому процессор прекращает выполнение команд в обычном режиме и, частично сохранив свое состояние, отвлекается на выполнение других действий, оказался настолько удобен, что зачастую разработчики процессоров используют его и для других целей. Хотя эти случаи и не относятся к операциям ввода-вывода, мы вынуждены упомянуть их здесь, для того чтобы их не путали с прерываниями. Похожим образом процессор обрабатывает исключительные ситуации и программные прерывания.

Для внешних прерываний характерны следующие особенности:

- Внешнее прерывание обнаруживается процессором между выполнением команд (или между итерациями в случае выполнения цепочечных команд).
- Процессор при переходе на обработку прерывания сохраняет часть своего состояния перед выполнением **следующей** команды.
- Прерывания происходят **асинхронно** с работой процессора и **непредсказуемо**, программист никоим образом не может предугадать, в каком именно месте работы программы произойдет прерывание.

Исключительные ситуации возникают во время выполнения процессором команды. К их числу относятся ситуации переполнения, деления на ноль, обращения к отсутствующей странице памяти (см. часть III) и т. д. Для исключительных ситуаций характерно следующее.

- Исключительные ситуации обнаруживаются процессором во время выполнения команд.
- Процессор при переходе на выполнение обработки исключительной ситуации сохраняет часть своего состояния перед выполнением текущей команды.
- Исключительные ситуации возникают **синхронно** с работой процессора, но **непредсказуемо** для программиста, если только тот специально не заставил процессор делить некоторое число на ноль.

Программные прерывания возникают после выполнения специальных команд, как правило, для выполнения привилегированных действий внутри системных вызовов. Программные прерывания имеют следующие свойства:

- Программное прерывание происходит в результате выполнения специальной команды.
- Процессор при выполнении программного прерывания сохраняет свое состояние перед выполнением **следующей** команды.
- Программные прерывания, естественно, возникают синхронно с работой процессора и **абсолютно предсказуемы** программистом.

Надо сказать, что реализация похожих механизмов обработки внешних прерываний, исключительных ситуаций и программных прерываний лежит целиком на совести разработчиков процессоров. Существуют вычислительные системы, где все три ситуации обрабатываются по-разному.

Прямой доступ к памяти (Direct Memory Access – DMA)

Использование механизма прерываний позволяет разумно загружать процессор в то время, когда устройство ввода-вывода занимается своей работой. Однако запись или чтение большого количества информации из адресного пространства ввода-вывода (например, с диска) приводят к большому количеству операций ввода-вывода, которые должен выполнять процессор. Для освобождения процессора от операций последовательного вывода данных из оперативной памяти или последовательного ввода в нее был предложен механизм прямого доступа внешних устройств к памяти – ПДП или Direct Memory Access – DMA. Давайте кратко рассмотрим, как работает этот механизм.

Для того чтобы какое-либо устройство, кроме процессора, могло записать информацию в память или прочитать ее из памяти, необходимо чтобы это устройство могло забрать у процессора управление локальной магистралью для выставления соответствующих сигналов на шины адреса, данных и управления. Для централизации эти обязанности обычно возлагаются не на каждое устройство в отдельности, а на специальный контроллер – контроллер прямого доступа к памяти. Контроллер прямого доступа к памяти имеет несколько спаренных линий – каналов DMA, которые могут подключаться к различным устройствам. Перед началом использования прямого доступа к памяти этот контроллер необходимо запрограммировать, записав в его порты информацию о том, какой канал или каналы предполагается задействовать, какие операции они будут совершать, какой адрес памяти является начальным для передачи информации и какое количество информации должно быть передано. Получив по одной из линий – каналов DMA сигнал запроса на передачу данных от внешнего устройства, контроллер по шине управления сообщает процессору о желании взять на себя управление локальной магистралью. Процессор, возможно, через некоторое время, необходимое для завершения его действий с магистралью, передает управление ею контроллеру DMA, известив его специальным сигналом. Контроллер DMA выставляет на адресную шину адрес памяти для передачи очередной порции информации и по второй линии канала прямого доступа к памяти сообщает устройству о готовности магистрали к передаче данных. После этого, используя шину данных и шину управления, контроллер DMA, устройство ввода-вывода и память осуществляют процесс обмена информацией. Затем контроллер прямого доступа к памяти извещает процессор о своем отказе от управления магистралью, и тот берет руководящие функции на себя. При передаче большого количества данных весь процесс повторяется циклически.

При прямом доступе к памяти процессор и контроллер DMA по очереди управляют локальной магистралью. Это, конечно, несколько

снижает производительность процессора, так как при выполнении некоторых команд или при чтении очередной порции команд во внутренний кэш он должен поджидать освобождения магистрали, но в целом производительность вычислительной системы существенно возрастает.

При подключении к системе нового устройства, которое умеет использовать прямой доступ к памяти, обычно необходимо программно или аппаратно задать номер канала DMA, к которому будет приписано устройство. В отличие от прерываний, где один номер прерывания мог соответствовать нескольким устройствам, каналы DMA всегда находятся в монопольном владении устройств.

Логические принципы организации ввода-вывода

Рассмотренные в предыдущем разделе физические механизмы взаимодействия устройств ввода-вывода с вычислительной системой позволяют понять, почему разнообразные внешние устройства легко могут быть добавлены в существующие компьютеры. Все, что необходимо сделать пользователю при подключении нового устройства, – это отобразить порты устройства в соответствующее адресное пространство, определить, какой номер будет соответствовать прерыванию, генерируемому устройством, и, если нужно, закрепить за устройством некоторый канал DMA. Для нормального функционирования hardware этого будет достаточно. Однако мы до сих пор ничего не сказали о том, как должна быть построена подсистема управления вводом-выводом в операционной системе для легкого и безболезненного добавления новых устройств, и какие функции вообще обычно на нее возлагаются.

Структура системы ввода-вывода

Если поручить неподготовленному пользователю сконструировать систему ввода-вывода, способную работать со всем множеством внешних устройств, то, скорее всего, он окажется в ситуации, в которой находились биологи и зоологи до появления трудов Линнея [Linnaes, 1789]. Все устройства разные, отличаются по выполняемым функциям и своим характеристикам, и кажется, что принципиально невозможно создать систему, которая без больших постоянных переделок позволяла бы охватывать все многообразие видов. Вот перечень лишь нескольких направлений (далеко не полный), по которым различаются устройства:

- Скорость обмена информацией может варьироваться в диапазоне от нескольких байтов в секунду (клавиатура) до нескольких гигабайтов в секунду (сетевые карты).

- Одни устройства могут использоваться несколькими процессами параллельно (являются разделяемыми), в то время как другие требуют монопольного захвата процессом.
- Устройства могут запоминать выведенную информацию для ее последующего ввода или не обладать этой функцией. Устройства, запоминающие информацию, в свою очередь, могут дифференцироваться по формам доступа к сохраненной информации: обеспечивать к ней последовательный доступ в жестко заданном порядке или уметь находить и передавать только необходимую порцию данных.
- Часть устройств умеет передавать данные только по одному байту последовательно (*символьные устройства*), а часть устройств умеет передавать блок байтов как единое целое (*блочные устройства*).
- Существуют устройства, предназначенные только для ввода информации, устройства, предназначенные только для вывода информации, а также устройства, которые могут выполнять и ввод, и вывод.

В области технического обеспечения удалось выделить несколько основных принципов взаимодействия внешних устройств с вычислительной системой, т. е. создать единый интерфейс для их подключения, возложив все специфические действия на контроллеры самих устройств. Тем самым конструкторы вычислительных систем переложили все хлопоты, связанные с подключением внешней аппаратуры, на разработчиков самой аппаратуры, заставляя их придерживаться определенного стандарта.

Похожий подход оказался продуктивным и в области программного подключения устройств ввода-вывода. Подобно тому как Линнею удалось заложить основы систематизации знаний о растительном и животном мире, разделив все живое в природе на относительно небольшое число классов и отрядов, мы можем разделить устройства на относительно небольшое число типов, отличающихся по набору операций, которые могут быть ими выполнены, считая все остальные различия несущественными. Мы можем затем специфицировать интерфейсы между ядром операционной системы, осуществляющим некоторую общую политику ввода-вывода, и программными частями, непосредственно управляющими устройствами, для каждого из таких типов. Более того, разработчики операционных систем получают возможность освободиться от написания и тестирования этих специфических программных частей, получивших название *драйверов*, передав эту деятельность производителям самих внешних устройств. Фактически мы приходим к использованию принципа уровневого или слоеного построения системы управления вводом-выводом для операционной системы (см. рис. 13.1).

Два нижних уровня этой слоеной системы составляет hardware: сами устройства, непосредственно выполняющие операции, и их контроллеры, служащие для организации совместной работы устройств и остальной

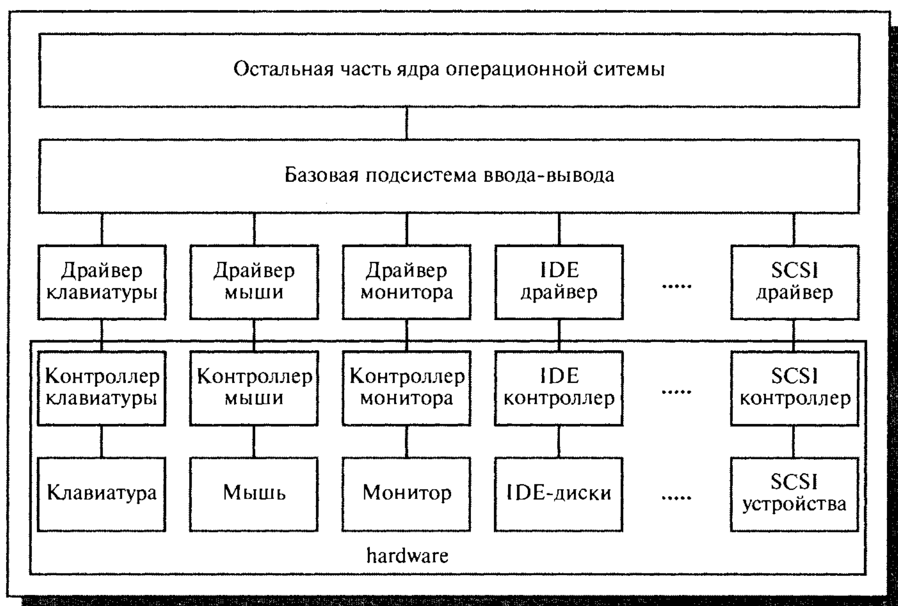


Рис. 13.1. Структура системы ввода-вывода

вычислительной системы. Следующий уровень составляют драйверы устройств ввода-вывода, скрывающие от разработчиков операционных систем особенности функционирования конкретных приборов и обеспечивающие четко определенный интерфейс между hardware и вышележащим уровнем – уровнем базовой подсистемы ввода-вывода, которая, в свою очередь, предоставляет механизм взаимодействия между драйверами и программной частью вычислительной системы в целом.

В последующих разделах мы подробнее рассмотрим организацию и функции набора драйверов и базовой подсистемы ввода-вывода.

Систематизация внешних устройств и интерфейс между базовой подсистемой ввода-вывода и драйверами

Как и система видов Линнея, система типов устройств является далеко не полной и не строго выдержанной. Устройства обычно принято разделять по преобладающему типу интерфейса на следующие виды:

- символьные (клавиатура, модем, терминал и т. п.);
- блочные (магнитные и оптические диски и ленты, и т. д.);
- сетевые (сетевые карты);
- все остальные (таймеры, графические дисплеи, телевизионные устройства, видеокамеры и т. п.).

Такое деление является весьма условным. В одних операционных системах сетевые устройства могут не выделяться в отдельную группу, в некоторых других – отдельные группы составляют звуковые устройства и видеоустройства и т. д. Некоторые группы в свою очередь могут разбиваться на подгруппы: подгруппа жестких дисков, подгруппа мышек, подгруппа принтеров. Нас такие детали не интересуют. Мы не ставим перед собой цель осуществить систематизацию всех возможных устройств, которые могут быть подключены к вычислительной системе. Единственное, для чего нам понадобится эта классификация, так это для иллюстрации того положения, что устройства могут быть разделены на группы по выполняемым ими функциям, и для понимания функций драйверов, и интерфейса между ними и базовой подсистемой ввода-вывода.

Для этого мы рассмотрим только две группы устройств: символьные и блочные. Как уже упоминалось в предыдущем разделе, символьные устройства – это устройства, которые умеют передавать данные только последовательно, байт за байтом, а блочные устройства – это устройства, которые могут передавать блок байтов как единое целое.

К символьным устройствам обычно относятся устройства ввода информации, которые спонтанно генерируют входные данные: клавиатура, мышь, модем, джойстик. К ним же относятся и устройства вывода информации, для которых характерно представление данных в виде линейного потока: принтеры, звуковые карты и т. д. По своей природе символьные устройства обычно умеют совершать две общие операции: ввести символ (байт) и вывести символ (байт) – `get` и `put`.

Для блочных устройств, таких как магнитные и оптические диски, ленты и т. п. естественными являются операции чтения и записи блока информации – `read` и `write`, а также, для устройств прямого доступа, операция поиска требуемого блока информации – `seek`.

Драйверы символьных и блочных устройств должны предоставлять базовой подсистеме ввода-вывода функции для осуществления описанных общих операций. Помимо общих операций, некоторые устройства могут выполнять операции специфические, свойственные только им – например, звуковые карты умеют увеличивать или уменьшать среднюю громкость звучания, дисплеи умеют изменять свою разрешающую способность. Для выполнения таких специфических действий в интерфейс между драйвером и базовой подсистемой ввода-вывода обычно входит еще одна функция, позволяющая непосредственно передавать драйверу устройства произвольную команду с произвольными параметрами, что позволяет задействовать любую возможность драйвера без изменения интерфейса. В операционной системе Unix такая функция получила название `ioctl` (от input-output control).

Помимо функций `read`, `write`, `seek` (для блочных устройств), `get`, `put` (для символьных устройств) и `ioctl`, в состав интерфейса обычно включают также следующие функции:

- Функцию инициализации или повторной инициализации работы драйвера и устройства – `open`.
- Функцию временного завершения работы с устройством (может, например, вызывать отключение устройства) – `close`.
- Функцию опроса состояния устройства (если по каким-либо причинам работа с устройством производится методом опроса его состояния, например, в операционных системах Windows NT и Windows 9x так построена работа с принтерами через параллельный порт) – `poll`.
- Функцию останова драйвера, которая вызывается при останове операционной системы или выгрузке драйвера из памяти, `halt`.

Существует еще ряд действий, выполнение которых может быть возложено на драйвер, но поскольку, как правило, это действия базовой подсистемы ввода-вывода, мы поговорим о них в следующем разделе. Приведенные выше названия функций, конечно, являются условными и могут меняться от одной операционной системы к другой, но действия, выполняемые драйверами, характерны для большинства операционных систем, и соответствующие функции присутствуют в интерфейсах к ним.

Функции базовой подсистемы ввода-вывода

Базовая подсистема ввода-вывода служит посредником между процессами вычислительной системы и набором драйверов. Системные вызовы для выполнения операций ввода-вывода трансформируются ею в вызовы функций необходимого драйвера устройства. Однако обязанности базовой подсистемы не сводятся к выполнению только действий трансляции общего системного вызова в обращение к частной функции драйвера. Базовая подсистема предоставляет вычислительной системе такие услуги, как поддержка блокирующихся, неблокирующихся и асинхронных системных вызовов, буферизация и кэширование входных и выходных данных, осуществление спулинга и монопольного захвата внешних устройств, обработка ошибок и прерываний, возникающих при операциях ввода-вывода, планирование последовательности запросов на выполнение этих операций. Давайте остановимся на этих услугах подробнее.

Блокирующиеся, неблокирующиеся и асинхронные системные вызовы

Все системные вызовы, связанные с осуществлением операций ввода-вывода, можно разбить на три группы по способам реализации взаимодействия процесса и устройства ввода-вывода.

- К первой, наиболее привычной для большинства программистов группе относятся блокирующиеся системные вызовы. Как следует из самого названия, применение такого вызова приводит к блокировке инициировавшего его процесса, т. е. процесс переводится операционной системой из состояния ***исполнение*** в состояние ***ожидание***. Завершив выполнение всех операций ввода-вывода, предписанных системным вызовом, операционная система переводит процесс из состояния ***ожидание*** в состояние ***готовность***. После того как процесс будет снова выбран для ***исполнения***, в нем произойдет окончательный возврат из системного вызова. Типичным для применения такого системного вызова является случай, когда процессу необходимо получить от устройства строго определенное количество данных, без которых он не может выполнять работу далее.
- Ко второй группе относятся *неблокирующиеся* системные вызовы. Их название не совсем точно отражает суть дела. В простейшем случае процесс, применивший неблокирующийся вызов, не переводится в состояние ***ожидание*** вообще. Системный вызов возвращается немедленно, выполнив предписанные ему операции ввода-вывода полностью, частично или не выполнив совсем, в зависимости от текущей ситуации (состояния устройства, наличия данных и т. д.). В более сложных ситуациях процесс может блокироваться, но условием его разблокирования является завершение всех необходимых операций или окончание некоторого промежутка времени. Типичным случаем применения неблокирующегося системного вызова может являться периодическая проверка на поступление информации с клавиатуры при выполнении трудоемких расчетов.
- К третьей группе относятся *асинхронные* системные вызовы. Процесс, использовавший асинхронный системный вызов, никогда в нем не блокируется. Системный вызов инициирует выполнение необходимых операций ввода-вывода и немедленно возвращается, после чего процесс продолжает свою регулярную деятельность. Об окончании завершения операции ввода-вывода операционная система впоследствии информирует процесс изменением значений некоторых переменных, передачей ему сигнала или сообщения или каким-либо иным способом. Необходимо четко понимать разницу между неблокирующимися и асинхронными вызовами. Неблокиру-

ющийся системный вызов для выполнения операции `read` вернется немедленно, но может прочитать запрошенное количество байтов, меньшее количество или вообще ничего. Асинхронный системный вызов для этой операции также вернется немедленно, но требуемое количество байтов рано или поздно будет прочитано в полном объеме.

Буферизация и кэширование

Под *буфером* обычно понимается некоторая область памяти для запоминания информации при обмене данных между двумя устройствами, двумя процессами или процессом и устройством. Обмен информацией между двумя процессами относится к области кооперации процессов, и мы подробно рассмотрели его организацию в соответствующей лекции. Здесь нас будет интересовать использование буферов в том случае, когда одним из участников обмена является внешнее устройство. Существует три причины, приводящие к использованию буферов в базовой подсистеме ввода-вывода:

- Первая причина буферизации – это разные скорости приема и передачи информации, которыми обладают участники обмена. Рассмотрим, например, случай передачи потока данных от клавиатуры к модему. Скорость, с которой поставляет информацию клавиатура, определяется скоростью набора текста человеком и обычно существенно меньше скорости передачи данных модемом. Для того чтобы не занимать модем на все время набора текста, делая его недоступным для других процессов и устройств, целесообразно накапливать введенную информацию в буфере или нескольких буферах достаточного размера и отсылать ее через модем после заполнения буферов.
- Вторая причина буферизации – это разные объемы данных, которые могут быть приняты или получены участниками обмена единовременно. Возьмем другой пример. Пусть информация поставляется модемом и записывается на жесткий диск. Помимо обладания разными скоростями совершения операций, модем и жесткий диск представляют собой устройства разного типа. Модем является символьным устройством и выдает данные байт за байтом, в то время как диск является блочным устройством и для проведения операции записи для него требуется накопить необходимый блок данных в буфере. Здесь также можно применять более одного буфера. После заполнения первого буфера модем начинает заполнять второй, одновременно с записью первого на жесткий диск. Поскольку скорость работы жесткого диска в тысячи раз больше, чем скорость работы модема, к моменту заполнения второго буфера операция записи первого будет

завершена, и модем снова сможет заполнять первый буфер одновременно с записью второго на диск.

- Третья причина буферизации связана с необходимостью копирования информации из приложений, осуществляющих ввод-вывод, в буфер ядра операционной системы и обратно. Допустим, что некоторый пользовательский процесс пожелал вывести информацию из своего адресного пространства на внешнее устройство. Для этого он должен выполнить системный вызов с обобщенным названием `write`, передав в качестве параметров адрес области памяти, где расположены данные, и их объем. Если внешнее устройство временно занято, то возможна ситуация, когда к моменту его освобождения содержимое нужной области окажется испорченным (например, при использовании асинхронной формы системного вызова). Чтобы избежать возникновения подобных ситуаций, проще всего в начале работы системного вызова скопировать необходимые данные в буфер ядра операционной системы, постоянно находящийся в оперативной памяти, и выводить их на устройство из этого буфера.

Под словом *кэш* (cache – «наличные»), этимологию которого мы не будем здесь рассматривать, обычно понимают область быстрой памяти, содержащую копию данных, расположенных где-либо в более медленной памяти, предназначенную для ускорения работы вычислительной системы. Мы с вами сталкивались с этим понятием при рассмотрении иерархии памяти. В базовой подсистеме ввода-вывода не следует смешивать два понятия — буферизацию и кэширование, хотя зачастую для выполнения этих функций отводится одна и та же область памяти. Буфер часто содержит единственный набор данных, существующий в системе, в то время как кэш по определению содержит копию данных, существующих где-нибудь еще. Например, буфер, используемый базовой подсистемой для копирования данных из пользовательского пространства процесса при выводе на диск, может в свою очередь применяться как кэш для этих данных, если операции модификации и повторного чтения данного блока выполняются достаточно часто.

Функции буферизации и кэширования не обязательно должны быть локализованы в базовой подсистеме ввода-вывода. Они могут быть частично реализованы в драйверах и даже в контроллерах устройств, скрытно по отношению к базовой подсистеме.

Спулинг и захват устройств

О понятии spooling мы говорили в первой лекции нашего курса, как о механизме, впервые позволившем совместить реальные операции ввода-вывода одного задания с выполнением другого задания. Теперь мы можем

определить это понятие более точно. Под словом spool мы подразумеваем буфер, содержащий входные или выходные данные для устройства, на котором следует избегать чередования его использования (возникновения interleaving – см. раздел «Interleaving, race condition и взаимоисключения» лекции 5) различными процессами. Правда, в современных вычислительных системах spool для ввода данных практически не используется, а в основном предназначен для накопления выходной информации.

Рассмотрим в качестве внешнего устройства принтер. Хотя принтер не может печатать информацию, поступающую одновременно от нескольких процессов, может оказаться желательным разрешить процессам совершать вывод на принтер параллельно. Для этого операционная система вместо передачи информации напрямую на принтер накапливает выводимые данные в буферах на диске, организованных в виде отдельного spool-файла для каждого процесса. После завершения некоторого процесса соответствующий ему spool-файл ставится в очередь для реальной печати. Механизм, обеспечивающий подобные действия, и получил название spooling.

В некоторых операционных системах вместо использования spooling для устранения race condition применяется механизм монопольного захвата устройств процессами. Если устройство свободно, то один из процессов может получить его в монопольное распоряжение. При этом все другие процессы при попытке осуществления операций над этим устройством будут либо блокированы (переведены в состояние ***ожидание***), либо получат информацию о невозможности выполнения операции до тех пор, пока процесс, захвативший устройство, не завершится или явно не сообщит операционной системе о своем отказе от его использования.

Обеспечение spooling и механизма захвата устройств является прерогативой базовой подсистемы ввода-вывода.

Обработка прерываний и ошибок

Если при работе с внешним устройством вычислительная система не пользуется методом опроса его состояния, а задействует механизм прерываний, то при возникновении прерывания, как мы уже говорили раньше, процессор, частично сохранив свое состояние, передает управление специальной программе обработки прерывания. Мы уже рассматривали действия операционной системы над процессами, происходящими при возникновении прерывания, в разделе «Переключение контекста» лекции 2, где после возникновения прерывания осуществлялись следующие действия: сохранение контекста, обработка прерывания, планирование использования процессора, восстановление контекста. Тогда мы обращали больше внимания на действия, связанные с сохранением и восстановлени-

ем контекста и планированием использования процессора. Теперь давайте подробнее остановимся на том, что скрывается за словами «обработка прерывания».

Одна и та же процедура обработки прерывания может применяться для нескольких устройств ввода-вывода (например, если эти устройства используют одну линию прерываний, идущую от них к контроллеру прерываний), поэтому первое действие собственно программы обработки состоит в определении того, какое именно устройство выдало прерывание. Зная устройство, мы можем выявить процесс, который инициировал выполнение соответствующей операции. Поскольку прерывание возникает как при удачном, так и при неудачном ее выполнении, следующее, что мы должны сделать, – это определить успешность завершения операции, проверив значение ***бита ошибки*** в регистре ***состояния*** устройства. В некоторых случаях операционная система может предпринять определенные действия, направленные на компенсацию возникшей ошибки. Например, в случае возникновения ошибки чтения с гибкого диска можно попробовать несколько раз повторить выполнение команды. Если компенсация ошибки невозможна, то операционная система впоследствии известит об этом процесс, запросивший выполнение операции (например, специальным кодом возврата из системного вызова). Если этот процесс был заблокирован до выполнения завершившейся операции, то операционная система переводит его в состояние ***готовность***. При наличии других неудовлетворенных запросов к освободившемуся устройству операционная система может инициировать выполнение следующего запроса, одновременно известив устройство, что прерывание обработано. На этом, собственно, обработка прерывания заканчивается, и система может приступать к планированию использования процессора.

Действия по обработке прерывания и компенсации возникающих ошибок могут быть частично переложены на плечи соответствующего драйвера. Для этого в состав интерфейса между драйвером и базовой подсистемой ввода-вывода добавляют еще одну функцию – функцию обработки прерывания `intr`.

Планирование запросов

При использовании неблокирующегося системного вызова может оказаться, что нужное устройство уже занято выполнением некоторых операций. В этом случае неблокирующийся вызов может немедленно вернуться, не выполнив запрошенных команд. При организации запроса на совершение операций ввода-вывода с помощью блокирующегося или асинхронного вызова занятость устройства приводит к необходимости постановки запроса в очередь к данному устройству. В результате с каж-

дым устройством оказывается связан список неудовлетворенных запросов процессов, находящихся в состоянии ***ожидания***, и запросов, выполняющихся в асинхронном режиме. Состояние ***ожидание*** расщепляется на набор очередей процессов, дожидающихся различных устройств ввода-вывода (или ожидающих изменения состояний различных объектов – семафоров, очередей сообщений, условных переменных в мониторах и т. д. – см. лекцию 6).

После завершения выполнения текущего запроса операционная система (по ходу обработки возникшего прерывания) должна решить, какой из запросов в списке должен быть удовлетворен следующим, и инициировать его исполнение. Точно так же, как для выбора очередного процесса на исполнение из списка готовых нам приходилось осуществлять краткосрочное планирование процессов, здесь нам необходимо осуществлять планирование применения устройств, пользуясь каким-либо алгоритмом этого планирования. Критерии и цели такого планирования мало отличаются от критериев и целей планирования процессов.

Задача планирования использования устройства обычно возлагается на базовую подсистему ввода-вывода, однако для некоторых устройств лучшие алгоритмы планирования могут быть тесно связаны с деталями их внутреннего функционирования. В таких случаях операция планирования переносится внутрь драйвера соответствующего устройства, так как эти детали скрыты от базовой подсистемы. Для этого в интерфейс драйвера добавляется еще одна специальная функция, которая осуществляет выбор очередного запроса, – функция `strategy`.

В следующем разделе мы рассмотрим некоторые алгоритмы планирования, связанные с удовлетворением запросов, на примере жесткого диска.

Алгоритмы планирования запросов к жесткому диску

Прежде чем приступить к непосредственному изложению самих алгоритмов, давайте вспомним внутреннее устройство жесткого диска и определим, какие параметры запросов мы можем использовать для планирования.

Строение жесткого диска и параметры планирования

Современный жесткий магнитный диск представляет собой набор круглых пластин, находящихся на одной оси и покрытых с одной или двух сторон специальным магнитным слоем (см. рис. 13.2). Около каждой рабочей поверхности каждой пластины расположены магнитные головки для чтения и записи информации. Эти головки присоединены к спе-

циальному рычагу, который может перемещать весь блок головок над поверхностями пластин как единое целое. Поверхности пластин разделены на концентрические кольца, внутри которых, собственно, и может храниться информация. Набор концентрических колец на всех пластинах для одного положения головок (т. е. все кольца, равноудаленные от оси) образует цилиндр. Каждое кольцо внутри цилиндра получило название дорожки (по одной или две дорожки на каждую пластину). Все дорожки делятся на равное число секторов. Количество дорожек, цилиндров и секторов может варьироваться от одного жесткого диска к другому в достаточно широких пределах. Как правило, сектор является минимальным объемом информации, которая может быть прочитана с диска за один раз.

При работе диска набор пластин вращается вокруг своей оси с высокой скоростью, подставляя по очереди под головки соответствующих дорожек все их сектора. Номер сектора, номер дорожки и номер цилиндра однозначно определяют положение данных на жестком диске и, наряду с типом совершаемой операции – чтение или запись, полностью характеризуют часть запроса, связанную с устройством, при обмене информацией в объеме одного сектора.

При планировании использования жесткого диска естественным параметром планирования является время, которое потребуется для выпол-

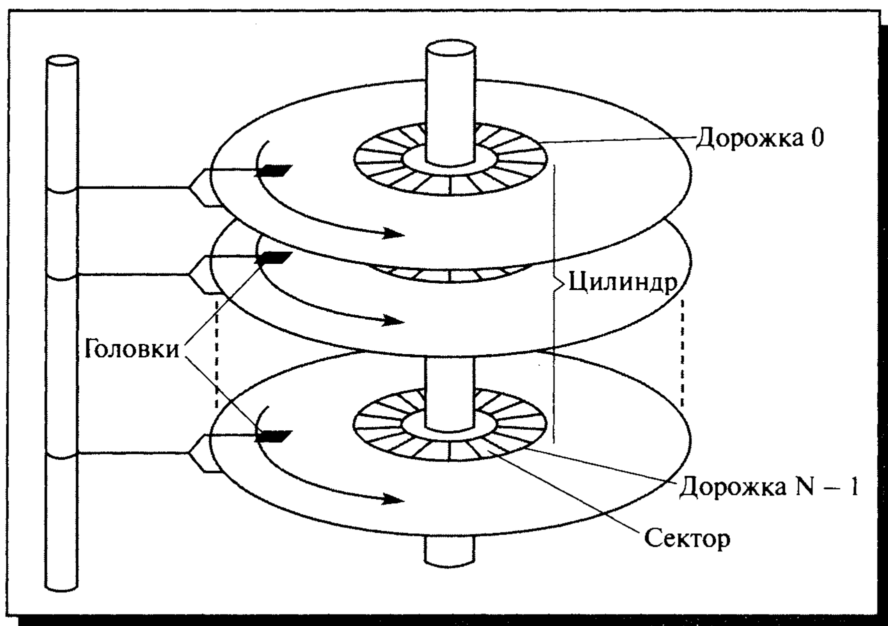


Рис. 13.2. Схема жесткого диска

нения очередного запроса. Время, необходимое для чтения или записи определенного сектора на определенной дорожке определенного цилиндра, можно разделить на две составляющие: время обмена информацией между магнитной головкой и компьютером, которое обычно не зависит от положения данных и определяется скоростью их передачи (transfer speed), и время, необходимое для позиционирования головки над заданным сектором, – время позиционирования (positioning time). Время позиционирования, в свою очередь, состоит из времени, необходимого для перемещения головок на нужный цилиндр, – времени поиска (seek time) и времени, которое требуется для того, чтобы нужный сектор довернулся под головку, – задержки на вращение (rotational latency). Времена поиска пропорциональны разнице между номерами цилиндров предыдущего и планируемого запросов, и их легко сравнивать. Задержка на вращение определяется довольно сложными соотношениями между номерами цилиндров и секторов предыдущего и планируемого запросов и скоростями вращения диска и перемещения головок. Без знания соотношения этих скоростей сравнение становится невозможным. Поэтому естественно, что набор параметров планирования сокращается до времени поиска различных запросов, определяемого текущим положением головки и номерами требуемых цилиндров, а разницей в задержках на вращение пренебрегают.

Алгоритм First Come First Served (FCFS)

Простейшим алгоритмом, к которому мы уже должны были привыкнуть, является алгоритм First Come First Served (FCFS) – первым пришел, первым обслужен. Все запросы организуются в очередь FIFO и обслуживаются в порядке поступления. Алгоритм прост в реализации, но может приводить к достаточно длительному общему времени обслуживания запросов. Рассмотрим пример. Пусть у нас на диске из 100 цилиндров (от 0 до 99) есть следующая очередь запросов: 23, 67, 55, 14, 31, 7, 84, 10 и головки в начальный момент находятся на 63-м цилиндре. Тогда положение головок будет меняться следующим образом:

$$63 \Rightarrow 23 \Rightarrow 67 \Rightarrow 55 \Rightarrow 14 \Rightarrow 31 \Rightarrow 7 \Rightarrow 84 \Rightarrow 10,$$

и всего головки переместятся на 329 цилиндров. Неэффективность алгоритма хорошо иллюстрируется двумя последними перемещениями с 7-го цилиндра через весь диск на 84-й цилиндр и затем опять через весь диск на цилиндр 10. Простая замена порядка двух последних перемещений ($7 \Rightarrow 10 \Rightarrow 84$) позволила бы существенно сократить общее время обслуживания запросов. Поэтому давайте перейдем к рассмотрению другого алгоритма.

Алгоритм Short Seek Time First (SSTF)

Как мы убедились, достаточно разумным является первоочередное обслуживание запросов, данные для которых лежат рядом с текущей позицией головок, а уж затем далеко отстоящих. Алгоритм Short Seek Time First (SSTF) – короткое время поиска первым – как раз и исходит из этой позиции. Для очередного обслуживания будем выбирать запрос, данные для которого лежат наиболее близко к текущему положению магнитных головок. Естественно, что при наличии равноудаленных запросов решение о выборе между ними может приниматься исходя из различных соображений, например по алгоритму FCFS. Для предыдущего примера алгоритм даст такую последовательность положений головок:

63⇒67⇒55⇒31⇒23⇒14⇒10⇒7⇒84,

и всего головки переместятся на 141-й цилиндр. Заметим, что наш алгоритм похож на алгоритм SJF планирования процессов, если за аналог оценки времени очередного CPU burst процесса выбирать расстояние между текущим положением головки и положением, необходимым для удовлетворения запроса. И точно так же, как алгоритм SJF, он может приводить к длительному откладыванию выполнения какого-либо запроса. Необходимо вспомнить, что запросы в очереди могут появляться в любой момент времени. Если у нас все запросы, кроме одного, постоянно группируются в области с большими номерами цилиндров, то этот один запрос может находиться в очереди неопределенно долго.

Точный алгоритм SJF являлся оптимальным для заданного набора процессов с заданными временами CPU burst. Очевидно, что алгоритм SSTF не является оптимальным. Если мы перенесем обслуживание запроса 67-го цилиндра в промежуток между запросами 7-го и 84-го цилиндров, мы уменьшим общее время обслуживания. Это наблюдение приводит нас к идее целого семейства других алгоритмов – алгоритмов сканирования.

Алгоритмы сканирования (SCAN, C-SCAN, LOOK, C-LOOK)

В простейшем из алгоритмов сканирования – SCAN – головки постоянно перемещаются от одного края диска до другого, по ходу дела обслуживая все встречающиеся запросы. По достижении другого края направление движения меняется, и все повторяется снова. Пусть в предыдущем примере в начальный момент времени головки двигаются в направлении уменьшения номеров цилиндров. Тогда мы и получим порядок обслуживания запросов, подсмотренный в конце предыдущего

раздела. Последовательность перемещения головок выглядит следующим образом:

63⇒55⇒31⇒23⇒14⇒10⇒7⇒0⇒67⇒84,

и всего головки переместятся на 147 цилиндров.

Если мы знаем, что обслужили последний попутный запрос в направлении движения головок, то мы можем не доходить до края диска, а сразу изменить направление движения на обратное:

63⇒55⇒31⇒23⇒14⇒10⇒7⇒67⇒84,

и всего головки переместятся на 133 цилиндра. Полученная модификация алгоритма SCAN получила название LOOK.

Допустим, что к моменту изменения направления движения головки в алгоритме SCAN, т. е. когда головка достигла одного из краев диска, у этого края накопилось большое количество новых запросов, на обслуживание которых будет потрачено достаточно много времени (не забываем, что надо не только перемещать головку, но еще и передавать прочитанные данные!). Тогда запросы, относящиеся к другому краю диска и поступившие раньше, будут ждать обслуживания несправедливо долго. Для сокращения времени ожидания запросов применяется другая модификация алгоритма SCAN – циклическое сканирование. Когда головка достигает одного из краев диска, она без чтения попутных запросов (иногда существенно быстрее, чем при выполнении обычного поиска цилиндра) перемещается на другой край, откуда вновь начинает движение в прежнем направлении. Для этого алгоритма, получившего название C-SCAN, последовательность перемещений будет выглядеть так:

63⇒55⇒31⇒23⇒14⇒10⇒7⇒0⇒99⇒84⇒67.

По аналогии с алгоритмом LOOK для алгоритма SCAN можно предложить и алгоритм C-LOOK для алгоритма C-SCAN

63⇒55⇒31⇒23⇒14⇒10⇒7⇒84⇒67.

Существуют и другие разновидности алгоритмов сканирования, и совсем другие алгоритмы, но мы на этом закончим, ибо было сказано: «И еще раз говорю: никто не обнимет необъятного».

Заключение

Функционирование любой вычислительной системы обычно сводится к выполнению двух видов работы: обработки информации и операций по осуществлению ее ввода-вывода. С точки зрения операционной системы «обработкой информации» являются только операции, совершаемые процессором над данными, находящимися в памяти на уровне иерархии не ниже чем оперативная память. Все остальное относится к «операциям ввода-вывода», т. е. к обмену информацией с внешними устройствами.

Несмотря на все многообразие устройств ввода-вывода, управление их работой и обмен информацией с ними строятся на относительно небольшом количестве принципов. Основными физическими принципами построения системы ввода-вывода являются следующие: возможность использования различных адресных пространств для памяти и устройств ввода-вывода; подключение устройств к системе через порты ввода-вывода, отображаемые в одно из адресных пространств; существование механизма прерывания для извещения процессора о завершении операций ввода-вывода; наличие механизма прямого доступа устройств к памяти, минуя процессор.

Механизм, подобный механизму прерываний, может использоваться также и для обработки исключений и программных прерываний, однако это целиком лежит на совести разработчиков вычислительных систем.

Для построения программной части системы ввода-вывода характерен «слоеный» подход. Для непосредственного взаимодействия с hardware используются драйверы устройств, скрывающие от остальной части операционной системы все особенности их функционирования. Драйверы устройств через жестко определенный интерфейс связаны с базовой подсистемой ввода-вывода, в обязанности которой входят: организация работы блокирующихся, неблокирующихся и асинхронных системных вызовов, буферизация и кэширование входных и выходных данных, осуществление spooling и монопольного захвата внешних устройств, обработка ошибок и прерываний, возникающих при операциях ввода-вывода, планирование последовательности запросов на выполнение этих операций. Доступ к базовой подсистеме ввода-вывода осуществляется посредством системных вызовов.

Часть функций базовой подсистемы может быть делегирована драйверам устройств и самим устройствам ввода-вывода.

Часть VI. Сети и сетевые операционные системы

Лекция 14. Сети и сетевые операционные системы

В лекции рассматриваются особенности взаимодействия процессов, выполняющихся на разных операционных системах, и вытекающие из этих особенностей функции сетевых частей операционных систем.

Ключевые слова: вычислительная сеть, сетевые операционные системы, распределенные операционные системы, удаленные процессы, локальные процессы, взаимодействие удаленных процессов, протокол, сетевой протокол, эталонная модель OSI/ISO, физический уровень, канальный уровень, сетевой уровень, транспортный уровень, уровень сеанса, уровень представления данных, прикладной уровень, проблема разрешения адресов, удаленный адрес процесса, локальный адрес процесса, сетевой порт, сокет (socket), Domain Name Service (DNS), алгоритмы маршрутизации, маршрутизация от источника данных, пошаговая маршрутизация, фиксированная маршрутизация, динамическая маршрутизация, простая маршрутизация, векторно-дистанционные алгоритмы, алгоритмы состояния связей, маршрутизатор, таблица маршрутизации, связь с установлением логического соединения, связь без установления логического соединения, датаграмма.

До сих пор в лекциях данного курса мы ограничивались рамками классических операционных систем, т. е. операционных систем, функционирующих на автономных однопроцессорных вычислительных машинах, которые к середине 80-х годов прошлого века составляли основу мирового парка вычислительной техники. Подчиняясь критериям повышения эффективности и удобства использования, вычислительные системы с этого времени, о чем мы уже упоминали в самой первой лекции, начинают бурно развиваться в двух направлениях: создание многопроцессорных компьютеров и объединение автономных систем в вычислительные сети.

Появление многопроцессорных компьютеров не оказывает существенного влияния на работу операционных систем. В многопроцессорной вычислительной системе изменяется содержание состояния ***исполнение***. В этом состоянии может находиться не один процесс, а несколько – по числу

процессоров. Соответственно изменяются и алгоритмы планирования. Наличие нескольких исполняющихся процессов требует более аккуратной реализации взаимоисключений при работе ядра. Но все эти изменения не являются изменениями идеологическими, не носят принципиального характера. Принципиальные изменения в многопроцессорных вычислительных комплексах затрагивают алгоритмический уровень, требуя разработки алгоритмов распараллеливания решения задач. Поскольку с точки зрения нашего курса многопроцессорные системы не внесли в развитие операционных систем что-либо принципиально новое, мы их рассматривать далее не будем.

По-другому обстоит дело с вычислительными сетями.

Для чего компьютеры объединяют в сети

Для чего вообще потребовалось объединять компьютеры в сети? Что привело к появлению сетей?

- Одной из главных причин стала необходимость совместного использования ресурсов (как физических, так и информационных). Если в организации имеется несколько компьютеров и эпизодически возникает потребность в печати какого-нибудь текста, то не имеет смысла покупать принтер для каждого компьютера. Гораздо выгоднее иметь один сетевой принтер для всех вычислительных машин. Аналогичная ситуация может возникать и с файлами данных. Зачем держать одинаковые файлы данных на всех компьютерах, поддерживая их когерентность, если можно хранить файл на одной машине, обеспечив к нему сетевой доступ со всех остальных?
- Второй причиной следует считать возможность ускорения вычислений. Здесь сетевые объединения машин успешно конкурируют с многопроцессорными вычислительными комплексами. Многопроцессорные системы, не затрагивая по существу строение операционных систем, требуют достаточно серьезных изменений на уровне hardware, что очень сильно повышает их стоимость. Во многих случаях можно добиться требуемой скорости вычислений параллельного алгоритма, используя не несколько процессоров внутри одного вычислительного комплекса, а несколько отдельных компьютеров, объединенных в сеть. Такие сетевые вычислительные кластеры часто имеют преимущество перед многопроцессорными комплексами в соотношении эффективность/стоимость.
- Следующая причина связана с повышением надежности работы вычислительной техники. В системах, где отказ может вызвать катастрофические последствия (атомная энергетика, космонавтика, авиация и т. д.), несколько вычислительных комплексов устанавливаются

в связи, дублируя друг друга. При выходе из строя основного комплекса его работу немедленно продолжает дублирующий.

- Наконец, последней по времени появления причиной (но для многих основной по важности) стала возможность применения вычислительных сетей для общения пользователей. Электронные письма практически заменили письма обычные, а использование вычислительной техники для организации электронных или телефонных разговоров уверенно вытесняет обычную телефонную связь.

Сетевые и распределенные операционные системы

В первой лекции мы говорили, что существует два основных подхода к организации операционных систем для вычислительных комплексов, связанных в сеть, – это сетевые и распределенные операционные системы. Необходимо отметить, что терминология в этой области еще не устоялась. В одних работах все операционные системы, обеспечивающие функционирование компьютеров в сети, называются распределенными, а в других, наоборот, сетевыми. Мы придерживаемся той точки зрения, что сетевые и распределенные системы являются принципиально различными.

В сетевых операционных системах для того, чтобы задействовать ресурсы другого сетевого компьютера, пользователи должны знать о его наличии и уметь это сделать. Каждая машина в сети работает под управлением своей локальной операционной системы, отличающейся от операционной системы автономного компьютера наличием дополнительных сетевых средств (программной поддержкой для сетевых интерфейсных устройств и доступа к удаленным ресурсам), но эти дополнения существенно не меняют структуру операционной системы.

Распределенная система, напротив, внешне выглядит как обычная автономная система. Пользователь не знает и не должен знать, где его файлы хранятся — на локальной или удаленной машине, и где его программы выполняются. Он может вообще не знать, подключен ли его компьютер к сети. Внутреннее строение распределенной операционной системы имеет существенные отличия от автономных систем.

Изучение строения распределенных операционных систем не входит в задачи нашего курса. Этому вопросу посвящены другие учебные курсы – Advanced operating systems, как называют их в англоязычных странах, или «Современные операционные системы», как принято называть их в России.

В этой лекции мы затронем вопросы, связанные с сетевыми операционными системами, а именно – какие изменения необходимо внести в классическую операционную систему для объединения компьютеров в сеть.

Взаимодействие удаленных процессов как основа работы вычислительных сетей

Все перечисленные выше цели объединения компьютеров в вычислительные сети не могут быть достигнуты без организации взаимодействия процессов на различных вычислительных системах. Будь то доступ к разделяемым ресурсам или общение пользователей через сеть – в основе всего этого лежит взаимодействие *удаленных* процессов, т. е. процессов, которые находятся под управлением физически разных операционных систем. Поэтому мы в своей работе сосредоточимся именно на вопросах кооперации таких процессов, в первую очередь выделив ее отличия от кооперации процессов в одной автономной вычислительной системе (кооперации *локальных* процессов), о которой мы говорили в лекциях 4, 5 и 6.

1) Изучая взаимодействие локальных процессов, мы разделили средства обмена информацией по объему передаваемых между ними данных и возможности влияния на поведение другого процесса на три категории: сигнальные, канальные и разделяемая память. На самом деле во всей этой систематизации присутствовала некоторая доля лукавства. Мы фактически классифицировали средства связи по виду интерфейса обращения к ним, в то время как реальной физической основой для всех средств связи в том или ином виде являлось разделение памяти. Семафоры представляют собой просто целочисленные переменные, лежащие в разделяемой памяти, к которым посредством системных вызовов, определяющих состав и содержание допустимых операций над ними, могут обращаться различные процессы. Очереди сообщений и pip'ы базируются на буферах ядра операционной системы, которые опять-таки с помощью системных вызовов доступны различным процессам. Иного способа реально передать информацию от процесса к процессу в автономной вычислительной системе просто не существует.

   Взаимодействие удаленных процессов принципиально отличается от ранее рассмотренных случаев. Общей памяти у различных компьютеров физически нет. Удаленные процессы могут обмениваться информацией, только передавая друг другу пакеты данных определенного формата (в виде последовательностей электрических или электромагнитных сигналов, включая световые) через некоторый физический канал связи или несколько таких каналов, соединяющих компьютеры. Поэтому в основе всех средств взаимодействия удаленных процессов лежит передача структурированных пакетов информации или сообщений.

2) При взаимодействии локальных процессов и процесс-отправитель информации, и процесс-получатель функционируют под управлением

одной и той же операционной системы. Эта же операционная система поддерживает и функционирование промежуточных накопителей данных при использовании непрямой адресации. Для организации взаимодействия процессы пользуются одними и теми же системными вызовами, присущими данной операционной системе, с одинаковыми интерфейсами. Более того, в автономной операционной системе передача информации от одного процесса к другому, независимо от используемого способа адресации, как правило (за исключением микроядерных операционных систем), происходит напрямую – без участия других процессов-посредников. Но даже и при наличии процессов-посредников все участники передачи информации находятся под управлением одной и той же операционной системы.

При организации сети, конечно, можно обеспечить прямую связь между всеми вычислительными комплексами, соединив каждый из них со всеми остальными посредством прямых физических линий связи или подключив все комплексы к общей шине (по примеру шин данных и адреса в компьютере). Однако такая сетевая топология не всегда возможна по ряду физических и финансовых причин. Поэтому во многих случаях информация между удаленными процессами в сети передается не напрямую, а через ряд процессов-посредников, «обитающих» на вычислительных комплексах, не являющихся компьютерами отправителя и получателя и работающих под управлением собственных операционных систем. Однако и при отсутствии процессов-посредников удаленные процесс-отправитель и процесс-получатель функционируют под управлением различных операционных систем, часто имеющих принципиально разное строение.

3) Вопросы надежности средств связи и способы ее реализации, рассмотренные нами в лекции 4, носили для случая локальных процессов скорее теоретический характер. Мы выяснили, что физической основой «общения» процессов на автономной вычислительной машине является разделяемая память. Поэтому для локальных процессов надежность передачи информации определяется надежностью ее передачи по шине данных и хранения в памяти машины, а также корректностью работы операционной системы. Для хороших вычислительных комплексов и операционных систем мы могли забыть про возможную ненадежность средств связи.

Для удаленных процессов вопросы, связанные с надежностью передачи данных, становятся куда более значимыми. Протяженные сетевые линии связи подвержены разнообразным физическим воздействиям, приводящим к искажению передаваемых по ним физических сигналов (помехи в эфире) или к полному отказу линий (мыши съели кабель). Даже при отсутствии внешних помех передаваемый сигнал

затухает по мере удаления от точки отправления, приближаясь по интенсивности к внутренним шумам линий связи. Промежуточные вычислительные комплексы сети, участвующие в доставке информации, не застрахованы от повреждений или внезапной перезагрузки операционной системы. Поэтому вычислительные сети должны организовываться исходя из предпосылок ненадежности доставки физических пакетов информации.

4) При организации взаимодействия локальных процессов каждый процесс (в случае прямой адресации) и каждый промежуточный объект для накопления данных (в случае непрямой адресации) должны были иметь уникальные идентификаторы – адреса – в рамках одной операционной системы. При организации взаимодействия удаленных процессов участники этого взаимодействия должны иметь уникальные адреса уже в рамках всей сети.

5) Физическая линия связи, соединяющая несколько вычислительных комплексов, является разделяемым ресурсом для всех процессов комплексов, которые хотят ее использовать. Если два процесса попытаются одновременно передать пакеты информации по одной и той же линии, то в результате интерференции физических сигналов, представляющих эти пакеты, произойдет взаимное искажение передаваемых данных. Для того чтобы избежать возникновения такой ситуации (race condition!) и обеспечить эффективную совместную работу вычислительных систем, должны выполняться условия взаимоисключения, прогресса и ограниченного ожидания при использовании общей линии связи, но уже не на уровне отдельных процессов операционных систем, а на уровне различных вычислительных комплексов в целом.

Давайте теперь, абстрагировавшись от физического уровня организации связи и не обращая внимания на то, какие именно физические средства – оптическое волокно, коаксиальный кабель, спутниковая связь и т. д. – лежат в основе объединения компьютеров в сеть, обсудим влияние перечисленных отличий на логические принципы организации взаимодействия удаленных процессов.

Основные вопросы логической организации передачи информации между удаленными процессами

К числу наиболее фундаментальных вопросов, связанных с логической организацией взаимодействия удаленных процессов, можно отнести следующие:

1) Как нужно соединять между собой различные вычислительные системы физическими линиями связи для организации взаимодействия удаленных процессов? Какими критериями при этом следует пользоваться?
2) Как избежать возникновения race condition при передаче информации различными вычислительными системами после их подключения к общей линии связи? Какие алгоритмы могут при этом применяться?
3) Какие виды интерфейсов могут быть предоставлены пользователю операционными системами для передачи информации по сети? Какие существуют модели взаимодействия удаленных процессов? Как процессы, работающие под управлением различных по своему строению операционных систем, могут общаться друг с другом?
4) Какие существуют подходы к организации адресации удаленных процессов? Насколько они эффективны?
5) Как организуется доставка информации от компьютера-отправителя к компьютеру-получателю через компьютеры-посредники? Как выбирается маршрут для передачи данных в случае разветвленной сетевой структуры, когда существует не один вариант следования пакетов данных через компьютеры-посредники?

Разумеется, степень важности этих вопросов во многом зависит от того, с какой точки зрения мы рассматриваем взаимодействие удаленных процессов. Системного программиста, в первую очередь, интересуют интерфейсы, предоставляемые операционными системами. Сетевого администратора больше будут занимать вопросы адресации процессов и выбора маршрута доставки данных. Проектировщика сетей в организации – способы соединения компьютеров и обеспечения корректного использования разделяемых сетевых ресурсов. Мы изучаем особенности строения и функционирования частей операционных систем, ответственных за взаимодействие удаленных процессов, а поэтому рассматриваемый перечень вопросов существенно сокращается.

Выбор способа соединения участников сетевого взаимодействия физическими линиями связи (количество и тип прокладываемых коммуникаций, какие именно устройства и как они будут соединять, т. е. топология сети) определяется проектировщиками сетей исходя из имеющихся средств, требуемой производительности и надежности взаимодействия. Все это не имеет отношения к функционированию операционных систем, является внешним по отношению к ним и в нашем курсе рассматриваться не будет.

В современных сетевых вычислительных комплексах решение вопросов организации взаимоисключений при использовании общей линии связи, как правило, также находится вне компетенции операционных систем и вынесено на физический уровень организации взаимодействия.

Ответственность за корректное использование коммуникаций возлагается на сетевые адаптеры, поэтому подобные проблемы мы тоже рассматривать не будем.

Из приведенного перечня мы с вами подробнее остановимся на решении вопросов, приведенных в пунктах 3–5.

Понятие протокола

Для описания происходящего в автономной операционной системе в лекции 2 было введено основополагающее понятие «процесс», на котором, по сути дела, базируется весь наш курс. Для того чтобы описать взаимодействие удаленных процессов и понять, какие функции и как должны выполнять дополнительные части сетевых операционных систем, отвечающих за такое взаимодействие, нам понадобится не менее фундаментальное понятие – *протокол*.

«Общение» локальных процессов напоминает общение людей, проживающих в одном городе. Если взаимодействующие процессы находятся под управлением различных операционных систем, то эта ситуация подобна общению людей, проживающих в разных городах и, возможно, в разных странах.

Каким образом два человека, находящиеся в разных городах, а тем более странах, могут обмениваться информацией? Для этого им обычно приходится прибегать к услугам соответствующих служб связи. При этом между службами связи различных городов (государств) должны быть заключены определенные соглашения, позволяющие корректно организовывать такой обмен. Если речь идет, например, о почтовых отправлениях, то в первую очередь необходимо договориться о том, что может представлять собой почтовое отправление, какой вид оно может иметь. Некоторые племена индейцев для передачи информации пользовались узелковыми письмами – поясами, к которым крепились веревочки с различным числом и формой узелков. Если бы такое письмо попало в современный почтовый ящик, то, пожалуй, ни одно отделение связи не догадалось бы, что это – письмо, и пояс был бы выброшен как ненужный хлам. Помимо формы представления информации необходима договоренность о том, какой служебной информацией должно снабжаться почтовое отправление (адрес получателя, срочность доставки, способ пересылки: поездом, авиацией, с помощью курьера и т. п.) и в каком формате она должна быть представлена. Адреса, например, в России и в США принято записывать по-разному. В России мы начинаем адрес со страны, далее указывается город, улица и квартира. В США все наоборот: сначала указывается квартира, затем улица и т. д. Конечно, даже при неправильной записи адреса письмо, скорее всего, дойдет до получателя, но можно себе представить

растерянность почтальона, пытающегося разгадать, что это за страна или город – «кв. 162»? Как видим, доставка почтового отправления из одного города (страны) в другой требует целого ряда соглашений между почтовыми ведомствами этих городов (стран).

Аналогичная ситуация возникает и при общении удаленных процессов, работающих под управлением разных операционных систем. Здесь процессы играют роль людей, проживающих в разных городах, а сетевые части операционных систем – роль соответствующих служб связи. Для того чтобы удаленные процессы могли обмениваться данными, необходимо, чтобы сетевые части операционных систем руководствовались определенными соглашениями, или, как принято говорить, поддерживали определенные *протоколы*. Термин «протокол» уже встречался нам в лекции 13, посвященной организации ввода-вывода в операционных системах, при обсуждении понятия «шина». Мы говорили, что понятие шины подразумевает не только набор проводников, но и набор жестко заданных протоколов, определяющий перечень сообщений, который может быть передан с помощью электрических сигналов по этим проводникам, т. е. в «протокол» мы вкладывали практически тот же смысл. В следующем разделе мы попытаемся дать более формализованное определение этого термина.

Необходимо отметить, что и локальные процессы при общении также должны руководствоваться определенными соглашениями или поддерживать определенные протоколы. Только в автономных операционных системах они несколько завуалированы. В роли таких протоколов выступают специальная последовательность системных вызовов при организации взаимодействия процессов и соглашения о параметрах системных вызовов.

Различные способы решения проблем 3–5, поднятых в предыдущем разделе, по существу, представляют собой различные соглашения, которых должны придерживаться сетевые части операционных систем, т. е. различные сетевые протоколы. Именно наличие сетевых протоколов позволяет организовать взаимодействие удаленных процессов.

При рассмотрении перечисленных выше проблем необходимо учитывать, с какими сетями мы имеем дело.

В литературе принято говорить о *локальных вычислительных сетях* (*LAN – Local Area Network*) и *глобальных вычислительных сетях* (*WAN – Wide Area Network*). Строгого определения этим понятиям обычно не дается, а принадлежность сети к тому или иному типу часто определяется взаимным расположением вычислительных комплексов, объединенных в сеть. Так, например, в большинстве работ к локальным сетям относят сети, состоящие из компьютеров одной организации, размещенные в пределах одного или нескольких зданий, а к глобальным сетям – сети, охватывающие все компьютеры в нескольких городах и более. Зачастую вводится дополнительный термин для описания сетей промежуточного масштаба –

муниципальных или *городских вычислительных сетей* (*MAN – Metropolitan Area Network*) – сетей, объединяющих компьютеры различных организаций в пределах одного города или одного городского района. Таким образом, упрощенно можно рассматривать глобальные сети как сети, состоящие из локальных и муниципальных сетей. А муниципальные сети, в свою очередь, могут состоять из нескольких локальных сетей. На самом деле деление сетей на локальные, глобальные и муниципальные обычно связано не столько с местоположением и принадлежностью вычислительных систем, соединенных сетью, сколько с различными подходами, применяемыми для решения поставленных вопросов в рамках той или иной сети, – с различными используемыми протоколами.

Многоуровневая модель построения сетевых вычислительных систем

Даже беглого взгляда на перечень проблем, связанных с логической организацией взаимодействия удаленных процессов, достаточно, чтобы понять, что построение сетевых средств связи – задача более сложная, чем реализация локальных средств связи. Поэтому обычно задачу создания таких средств решают по частям, применяя уже неоднократно упоминавшийся нами «слоеный», или многоуровневый, подход.

Как уже отмечалось при обсуждении «слоеного» строения операционных систем на первой лекции, при таком подходе уровень N системы предоставляет сервисы уровню N + 1, пользуясь в свою очередь только сервисами уровня N – 1. Следовательно, каждый уровень может взаимодействовать непосредственно только со своими соседями, руководствуясь четко закрепленными соглашениями – вертикальными протоколами, которые принято называть интерфейсами.

Самым нижним уровнем в слоеных сетевых вычислительных системах является уровень, на котором реализуется реальная физическая связь между двумя узлами сети. Из предыдущего раздела следует, что для обеспечения обмена физическими сигналами между двумя различными вычислительными системами необходимо, чтобы эти системы поддерживали определенный протокол физического взаимодействия – горизонтальный протокол.

На самом верхнем уровне находятся пользовательские процессы, которые инициируют обмен данными. Количество и функции промежуточных уровней варьируются от одной системы к другой. Вернемся к нашей аналогии с пересылкой почтовых отправлений между людьми, проживающими в разных городах, правда, с порядком их пересылки несколько отличным от привычного житейского порядка. Рассмотрим в качестве пользовательских процессов руководителей различных организаций, же-

лающих обменяться письмами. Руководитель (пользовательский процесс) готовит текст письма (данные) для пересылки в другую организацию. Затем он отдает его своему секретарю (совершает системный вызов – обращение к нижестоящему уровню), сообщая, кому и куда должно быть направлено письмо. Секретарь снимает с него копию и выясняет адрес организации. Далее идет обращение к нижестоящему уровню, допустим, письмо направляется в канцелярию. Здесь оно регистрируется (ему присваивается порядковый номер), один экземпляр запечатывается в конверт, на котором указывается, кому и куда адресовано письмо, впечатывается адрес отправителя. Копия остается в канцелярии, а конверт отправляется на почту (переход на следующий уровень). На почте наклеиваются марки и делаются другие служебные пометки, определяется способ доставки корреспонденции и т. д. Наконец, поездом, самолетом или курьером (физический уровень!) письмо следует в другой город, в котором все уровни проходятся в обратном порядке. Пользовательский уровень (руководитель) после подготовки исходных данных и инициации процесса взаимодействия далее судьбой почтового отправления не занимается. Все остальное (включая, быть может, проверку его доставки и посылку копии в случае утери) выполняют нижестоящие уровни.

Заметим, что на каждом уровне взаимодействия в городе отправителя исходные данные (текст письма) обрастают дополнительной служебной информацией. Соответствующие уровни почтовой службы в городе получателя должны уметь понимать эту служебную информацию. Таким образом, для одинаковых уровней в различных городах необходимо следование специальным соглашениям – поддержка определенных горизонтальных протоколов.

Точно так же в сетевых вычислительных системах все их одинаковые уровни, лежащие выше физического, виртуально обмениваются данными посредством горизонтальных протоколов. Наличие такой виртуальной связи означает, что уровень N компьютера 2 должен получить ту же самую информацию, которая была отправлена уровнем N компьютера 1. Хотя в реальности эта информация должна была сначала дойти сверху вниз до уровня 1 компьютера 1, затем передана уровню 1 компьютера 2 и только после этого доставлена снизу вверх уровню N этого компьютера.

Формальный перечень правил, определяющих последовательность и формат сообщений, которыми обмениваются сетевые компоненты различных вычислительных систем, лежащие на одном уровне, мы и будем называть сетевым протоколом.

Всю совокупность вертикальных и горизонтальных протоколов (интерфейсов и сетевых протоколов) в сетевых системах, построенных по «слоеному» принципу, достаточную для организации взаимодействия удаленных процессов, принято называть *семейством протоколов* или *сте-*

ком протоколов. Сети, построенные на основе разных стеков протоколов, могут быть объединены между собой с использованием вычислительных устройств, осуществляющих трансляцию из одного стека протоколов в другой, причем на различных уровнях слоеной модели.

Эталоном многоуровневой схемы построения сетевых средств связи считается семиуровневая модель открытого взаимодействия систем (Open System Interconnection – OSI), предложенная Международной организацией Стандартов (International Standard Organization – ISO) и получившая сокращенное наименование OSI/ISO (см. рис. 14.1).

Давайте очень кратко опишем, какие функции выполняют различные уровни модели OSI/ISO [Олифер, 2001]:

- Уровень 1 – *физический*. Этот уровень связан с работой hardware. На нем определяются физические аспекты передачи информации по линиям связи, такие как: напряжения, частоты, природа передающей среды, способ передачи двоичной информации по физическому носителю, вплоть до размеров и формы используемых разъемов. В компьютерах за поддержку физического уровня обычно отвечает сетевой адаптер.
- Уровень 2 – *канальный*. Этот уровень отвечает за передачу данных по физическому уровню без искажений между непосредственно связанными узлами сети. На нем формируются физические пакеты данных для реальной доставки по физическому уровню. Протоколы канального уровня реализуются совместно сетевыми адаптерами и их драйверами (понятие драйвера рассматривалось в лекции 13).
- Уровень 3 – *сетевой*. Сетевой уровень несет ответственность за доставку информации от узла-отправителя к узлу-получателю. На этом уровне частично решаются вопросы адресации, осуществляется выбор маршрутов следования пакетов данных, решаются вопросы стыковки сетей, а также управление скоростью передачи информации для предотвращения перегрузок в сети.
- Уровень 4 – *транспортный*. Регламентирует передачу данных между удаленными процессами. Обеспечивает доставку информации вышестоящим уровнем с необходимой степенью надежности, компенсируя, быть может, ненадежность нижестоящих уровней, связанную с искажением и потерей данных или доставкой пакетов в неправильном порядке. Наряду с сетевым уровнем может управлять скоростью передачи данных и частично решать проблемы адресации.
- Уровень 5 – *сеансовый*. Координирует взаимодействие связывающихся процессов. Основная задача – предоставление средств синхронизации взаимодействующих процессов. Такие средства синхронизации позволяют создавать контрольные точки при передаче больших объемов информации. В случае сбоя в работе сети передачу

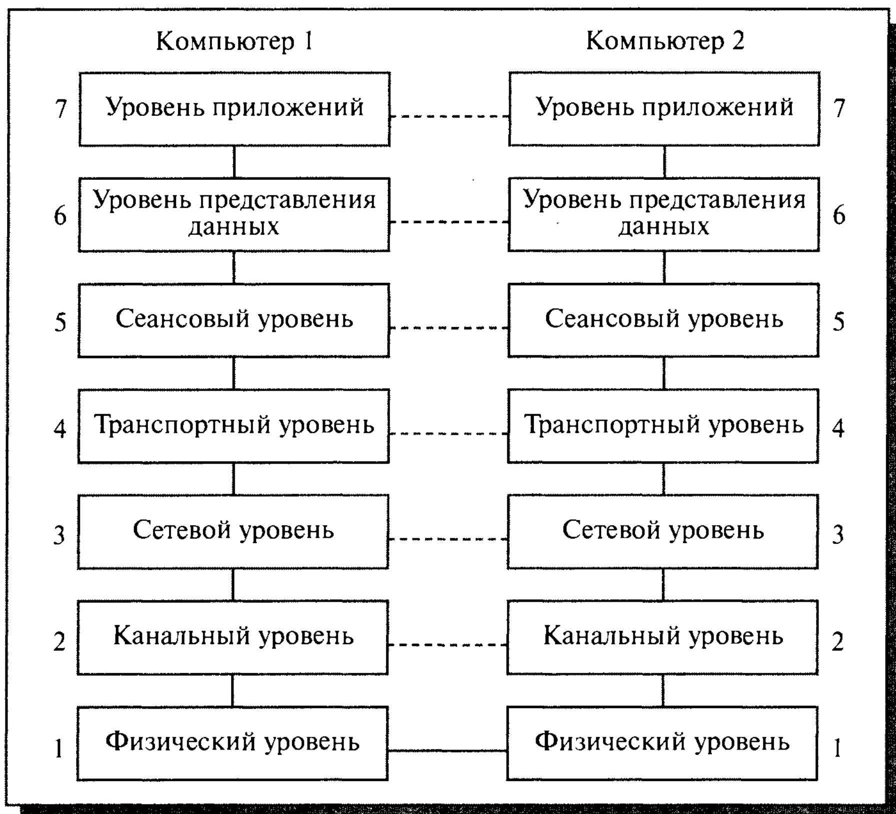


Рис. 14.1. Семиуровневая эталонная модель OSI/ISO

данных можно возобновить с последней контрольной точки, а не начинать заново.

- Уровень 6 – уровень *представления данных*. Отвечает за форму представления данных, перекодирует текстовую и графическую информацию из одного формата в другой, обеспечивает ее сжатие и распаковку, шифрование и декодирование.
- Уровень 7 – *прикладной*. Служит для организации интерфейса между пользователем и сетью. На этом уровне реализуются такие сервисы, как удаленная передача данных, удаленный терминальный доступ, почтовая служба и работа во Всемирной паутине (web-браузеры).

Надо отметить, что к приведенной эталонной модели большинство практиков относится без излишнего пиетета. Эта модель не предвосхитила появления различных семейств протоколов, таких как, например, семейство протоколов TCP/IP, а наоборот, была создана под их влиянием. Ее

не следует рассматривать как готовый оптимальный чертеж для создания любого сетевого средства связи. Наличие некоторой функции на определенном уровне не гарантирует, что это ее наилучшее место, некоторые функции (например, коррекция ошибок) дублируются на нескольких уровнях, да и само деление на 7 уровней носит отчасти произвольный характер. Хотя в конце концов были созданы работающие реализации этой модели, но наиболее распространенные семейства протоколов лишь до некоторой степени согласуются с ней. Как отмечено в книге [Таненбаум, 2002], она больше подходит для реализации телефонных, а не вычислительных сетей. Ценность предложенной эталонной модели заключается в том, что она показывает направление, в котором должны двигаться разработчики новых вычислительных сетей.

Проблемы 3–5, перечисленные в разделе «Основные вопросы логической организации передачи информации между удаленными процессами», относятся в основном к сетевому и транспортному уровням эталонной модели и, соответственно, решаются на уровне сетевых и транспортных протоколов. Давайте приступим, наконец, к их рассмотрению.

Проблемы адресации в сети

Любой пакет информации, передаваемый по сети, должен быть снабжен адресом получателя. Если взаимодействие подразумевает двустороннее общение, то в пакет следует также включить и адрес отправителя. В лекции 4 мы описали один из протоколов организации надежной связи с использованием контрольных сумм, нумерации пакетов и подтверждения получения неискаженного пакета в правильном порядке. Для отправки подтверждений обратный адрес также следует включать в пересылаемый пакет. Таким образом, практически каждый сетевой пакет информации должен быть снабжен адресом получателя и адресом отправителя. Как могут выглядеть такие адреса?

Несколько раньше, обсуждая отличия взаимодействия удаленных процессов от взаимодействия локальных процессов, мы говорили, что удаленные адресаты должны обладать уникальными адресами уже не в пределах одного компьютера, а в рамках всей сети. Существует два подхода к наделению объектов такими сетевыми адресами: одноуровневый и двухуровневый.

Одноуровневые адреса

В небольших компьютерных сетях можно построить одноуровневую систему адресации. При таком подходе каждый процесс, желающий

стать участником удаленного взаимодействия (при прямой адресации), и каждый объект, для такого взаимодействия предназначенный (при непрямой адресации), получают по мере необходимости собственные адреса (символьные или числовые), а сами вычислительные комплексы, объединенные в сеть, никаких самостоятельных адресов не имеют. Подобный метод требует довольно сложного протокола обеспечения уникальности адресов. Вычислительный комплекс, на котором запускается взаимодействующий процесс, должен запросить все компьютеры сети о возможности присвоения процессу некоторого адреса. Только после получения от них согласия процессу может быть назначен адрес. Поскольку процесс, посылающий данные другому процессу, не может знать, на каком компоненте сети находится процесс-адресат, передаваемая информация должна быть направлена всем компонентам сети (так называемое *широковещательное сообщение – broadcast message*), проанализирована ими и либо отброшена (если процесса-адресата на данном компьютере нет), либо доставлена по назначению. Так как все данные постоянно передаются от одного комплекса ко всем остальным, такую одноуровневую схему обычно применяют только в локальных сетях с прямой физической связью всех компьютеров между собой (например, в сети NetBIOS на базе Ethernet), но она является существенно менее эффективной, чем двухуровневая схема адресации.

Двухуровневые адреса

При двухуровневой адресации полный сетевой адрес процесса или промежуточного объекта для хранения данных складывается из двух частей – адреса вычислительного комплекса, на котором находится процесс или объект в сети (удаленного адреса), и адреса самого процесса или объекта на этом вычислительном комплексе (локального адреса). Уникальность полного адреса будет обеспечиваться уникальностью удаленного адреса для каждого компьютера в сети и уникальностью локальных адресов объектов на компьютере. Давайте подробнее рассмотрим проблемы, возникающие для каждого из компонентов полного адреса.

Удаленная адресация и разрешение адресов

Инициатором связи процессов друг с другом всегда является человек, будь то программист или обычный пользователь. Как мы неоднократно отмечали в лекциях, человеку свойственно думать словами, он легче воспринимает символьную информацию. Поэтому очевидно, что каждая машина в сети получает символьное, часто даже содержательное

имя. Компьютер не разбирается в смысловом содержании символов, ему проще оперировать числами, желательно одного и того же формата, которые помещаются, например, в 4 байт или в 16 байт. Поэтому каждый компьютер в сети для удобства работы вычислительных систем получает числовой адрес. Возникает проблема отображения пространства символьных имен (или адресов) вычислительных комплексов в пространство их числовых адресов. Эта проблема получила наименование *проблемы разрешения адресов*.

С подобными задачами мы уже сталкивались, обсуждая организацию памяти в вычислительных системах (отображение имен переменных в их адреса в процессе компиляции и редактирования связей) и организацию файловых систем (отображение имен файлов в их расположении на диске). Посмотрим, как она может быть решена в сетевом варианте.

Первый способ решения заключается в том, что на каждом сетевом компьютере создается файл, содержащий имена всех машин, доступных по сети, и их числовые эквиваленты. Обращаясь к этому файлу, операционная система легко может перевести символьный удаленный адрес в числовую форму. Такой подход использовался на заре эпохи глобальных сетей и применяется в изолированных локальных сетях в настоящее время. Действительно, легко поддерживать файл соответствий в корректном виде, внося в него необходимые изменения, когда общее число сетевых машин не превышает нескольких десятков. Как правило, изменения вносятся на некотором выделенном административном вычислительном комплексе, откуда затем обновленный файл рассылается по всем компонентам сети.

В современной сетевой паутине этот подход является неприемлемым. Дело даже не в размерах подобного файла, а в частоте требуемых обновлений и в огромном количестве рассылок, что может полностью подорвать производительность сети. Проблема состоит в том, что добавление или удаление компонента сети требует внесения изменений в файлы на всех сетевых машинах. Второй метод разрешения адресов заключается в частичном распределении информации о соответствии символьных и числовых адресов по многим комплексам сети, так что каждый из этих комплексов содержит лишь часть полных данных. Он же определяет и правила построения символических имен компьютеров.

Один из таких способов, используемый в Internet, получил английское наименование *Domain Name Service* или сокращенно DNS. Эта аббревиатура широко используется и в русскоязычной литературе. Давайте рассмотрим данный метод подробнее.

Организуем логически все компьютеры сети в некоторую древовидную структуру, напоминающую структуру директорий файловых систем, в которых отсутствует возможность организации жестких и мягких связей и

нет пустых директорий. Будем рассматривать все компьютеры, входящие во Всемирную сеть, как область самого низкого ранга (аналог корневой директории в файловой системе) – ранга 0. Разобьем все множество компьютеров области на какое-то количество подобластей (domains). При этом некоторые подобласти будут состоять из одного компьютера (аналоги регулярных файлов в файловых системах), а некоторые – более чем из одного компьютера (аналоги директорий в файловых системах). Каждую подобласть будем рассматривать как область более высокого ранга. Присвоим подобластям собственные имена таким образом, чтобы в рамках разбиваемой области все они были уникальны. Повторим такое разбиение рекурсивно для каждой области более высокого ранга, которая состоит более чем из одного компьютера, несколько раз, пока при последнем разбиении в каждой подобласти не окажется ровно по одному компьютеру. Глубина рекурсии для различных областей одного ранга может быть разной, но обычно в целом ограничиваются 3-5 разбиениями, начиная от ранга 0.

В результате мы получим дерево, неименованной вершиной которого является область, объединяющая все компьютеры, входящие во Всемирную сеть, именованными терминальными узлами – отдельные компьютеры (точнее – подобласти, состоящие из отдельных компьютеров), а именованными нетерминальными узлами – области различных рангов. Используем полученную структуру для построения имен компьютеров, подобно тому как мы поступали при построении полных имен файлов в структуре директорий файловой системы. Только теперь, двигаясь от корневой вершины к терминальному узлу – отдельному компьютеру, будем вести запись имен подобластей справа налево и отделять имена друг от друга с помощью символа «.».

Допустим, некоторая подобласть, состоящая из одного компьютера, получила имя ***serv***, она входит в подобласть, объединяющую все компьютеры некоторой лаборатории, с именем ***crec***. Та, в свою очередь, входит в подобласть всех компьютеров Московского физико-технического института с именем ***mipt***, которая включается в область ранга 1 всех компьютеров России с именем ***ru***. Тогда имя рассматриваемого компьютера во Всемирной сети будет ***serv.crec.mipt.ru***. Аналогичным образом можно именовать и подобласти, состоящие более чем из одного компьютера.

В каждой полученной именованной области, состоящей более чем из одного узла, выберем один из компьютеров и назначим его ответственным за эту область – сервером DNS. Сервер DNS знает числовые адреса серверов DNS для подобластей, входящих в его зону ответственности, или числовые адреса отдельных компьютеров, если такая подобласть включает в себя только один компьютер. Кроме того, он также знает числовой адрес сервера DNS, в зону ответственности которого входит рассматриваемая область (если это не область ранга 1), или числовые адреса

всех серверов DNS ранга 1 (в противном случае). Отдельные компьютеры всегда знают числовые адреса серверов DNS, которые непосредственно за них отвечают.

Рассмотрим теперь, как процесс на компьютере ***serv.crec.mipt.ru*** может узнать числовой адрес компьютера ***ssp.brown.edu***. Для этого он обращается к своему DNS-серверу, отвечающему за область ***crec.mipt.ru***, и передает ему нужный адрес в символьном виде. Если этот DNS-сервер не может сразу представить необходимый числовой адрес, он передает запрос DNS-серверу, отвечающему за область ***mipt.ru***. Если и тот не в силах самостоятельно справиться с проблемой, он перенаправляет запрос серверу DNS, отвечающему за область 1-го ранга ***ru***. Этот сервер может обратиться к серверу DNS, обслуживающему область 1-го ранга ***edu***, который, наконец, затребует информацию от сервера DNS области ***brown.edu***, где должен быть нужный числовой адрес. Полученный числовой адрес по всей цепи серверов DNS в обратном порядке будет передан процессу, направившему запрос (см. рис. 14.2).

В действительности, каждый сервер DNS имеет достаточно большой кэш, содержащий адреса серверов DNS для всех последних запросов. Поэтому реальная схема обычно существенно проще, из приведенной цепочки общения DNS-серверов выпадают многие звенья за счет обращения напрямую.

Рассмотренный способ разрешения адресов позволяет легко добавлять компьютеры в сеть и исключать их из сети, так как для этого необходимо внести изменения только на DNS-сервере соответствующей области.

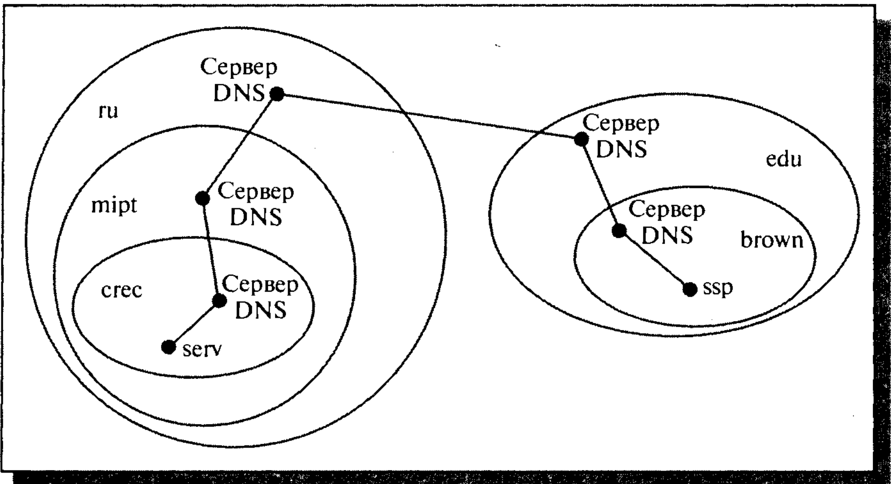


Рис. 14.2. Пример разрешения имен с использованием DNS-серверов

Если DNS-сервер, отвечающий за какую-либо область, выйдет из строя, то может оказаться невозможным разрешение адресов для всех компьютеров этой области. Поэтому обычно назначается не один сервер DNS, а два – основной и запасной. В случае выхода из строя основного сервера его функции немедленно начинает выполнять запасной.

В реальных сетевых вычислительных системах обычно используется комбинация рассмотренных подходов. Для компьютеров, с которыми чаще всего приходится устанавливать связь, в специальном файле хранится таблица соответствий символьных и числовых адресов. Все остальные адреса разрешаются с использованием служб, аналогичных службе DNS. Способ построения удаленных адресов и методы разрешения адресов обычно определяются протоколами сетевого уровня эталонной модели.

Мы разобрались с проблемой удаленных адресов и знаем, как получить числовой удаленный адрес нужного нам компьютера. Давайте рассмотрим теперь проблему адресов локальных: как нам задать адрес процесса или объекта для хранения данных на удаленном компьютере, который в конечном итоге и должен получить переданную информацию.

Локальная адресация. Понятие порта

Во второй лекции мы говорили, что каждый процесс, существующий в данный момент в вычислительной системе, уже имеет собственный уникальный номер – PID. Но этот номер неудобно использовать в качестве локального адреса процесса при организации удаленной связи. Номер, который получает процесс при рождении, определяется моментом его запуска, предысторией работы вычислительного комплекса и является в значительной степени случайным числом, изменяющимся от запуска к запуску. Представьте себе, что адресат, с которым вы часто переписываетесь, постоянно переезжает с место на место, меняя адреса, так что, посылая очередное письмо, вы не можете с уверенностью сказать, где он сейчас проживает, и поймете все неудобство использования идентификатора процесса в качестве его локального адреса. Все сказанное выше справедливо и для идентификаторов промежуточных объектов, использующихся при локальном взаимодействии процессов в схемах с непрямой адресацией.

Для локальной адресации процессов и промежуточных объектов при удаленной связи обычно организуется новое специальное адресное пространство, например представляющее собой ограниченный набор положительных целочисленных значений или множество символических имен, аналогичных полным именам файлов в файловых системах. Каждый процесс, желающий принять участие в сетевом взаимодействии, после рождения закрепляет за собой один или несколько адресов в этом адресном пространстве. Каждому промежуточному объекту при его созда-

нии присваивается свой адрес из этого адресного пространства. При этом удаленные пользователи могут заранее договориться о том, какие именно адреса будут зарезервированы для данного процесса, независимо от времени его старта, или для данного объекта, независимо от момента его создания. Подобные адреса получили название *портов*, по аналогии с портами ввода-вывода.

Необходимо отметить, что в системе может существовать несколько таких адресных пространств для различных способов связи. При получении данных от удаленного процесса операционная система смотрит, на какой порт и для какого способа связи они были отправлены, определяет процесс, который заявил этот порт в качестве своего адреса, или объект, которому присвоен данный адрес, и доставляет полученную информацию адресату. Виды адресного пространства портов (т. е. способы построения локальных адресов) определяются, как правило, протоколами транспортного уровня эталонной модели.

Полные адреса. Понятие сокета (socket)

Таким образом, полный адрес удаленного процесса или промежуточного объекта для конкретного способа связи с точки зрения операционных систем определяется парой адресов: <числовой адрес компьютера в сети, порт>. Подобная пара получила наименование socket (в переводе – «гнездо» или, как стали писать в последнее время, сокет), а сам способ их использования – *организация связи с помощью сокетов*. В случае непрямой адресации с использованием промежуточных объектов сами эти объекты также принято называть сокетами. Поскольку разные протоколы транспортного уровня требуют разных адресных пространств портов, то для каждой пары надо указывать, какой транспортный протокол она использует, – говорят о разных типах сокетов.

В современных сетевых системах числовой адрес обычно получает не сам вычислительный комплекс, а его сетевой адаптер, с помощью которого комплекс подключается к линии связи. При наличии нескольких сетевых адаптеров для разных линий связи один и тот же вычислительный комплекс может иметь несколько числовых адресов. В таких системах полные адреса удаленного адресата (процесса или промежуточного объекта) задаются парами <числовой адрес сетевого адаптера, порт> и требуют доставки информации через указанный сетевой адаптер.

Проблемы маршрутизации в сетях

При наличии прямой линии связи между двумя компьютерами обычно не возникает вопросов о том, каким именно путем должна

быть доставлена информация. Но, как уже упоминалось, одно из отличий взаимодействия удаленных процессов от взаимодействия процессов локальных состоит в использовании в большинстве случаев процессов-посредников, расположенных на вычислительных комплексах, не являющихся комплексами отправителя и получателя. В сложных топологических схемах организации сетей информация между двумя компьютерами может передаваться по различным путям. Возникает вопрос: как организовать работу операционных систем на комплексах-участниках связи (это могут быть конечные или промежуточные комплексы) для определения маршрута передачи данных? По какой из нескольких линий связи (или через какой сетевой адаптер) нужно отправить пакет информации? Какие протоколы маршрутизации возможны? Существует два принципиально разных подхода к решению этой проблемы: маршрутизация от источника передачи данных и одношаговая маршрутизация.

- *Маршрутизация от источника передачи данных.* При маршрутизации от источника данных полный маршрут передачи пакета по сети формируется на компьютере-отправителе в виде последовательности числовых адресов сетевых адаптеров, через которые должен пройти пакет, чтобы добраться до компьютера-получателя, и целиком включается в состав этого пакета. В этом случае промежуточные компоненты сети при определении дальнейшего направления движения пакета не принимают самостоятельно никаких решений, а следуют указаниям, содержащимся в пакете.
- *Одношаговая маршрутизация.* При одношаговой маршрутизации каждый компонент сети, принимающий участие в передаче информации, самостоятельно определяет, какому следующему компоненту, находящемуся в зоне прямого доступа, она должна быть отправлена. Решение принимается на основании анализа содержащегося в пакете адреса получателя. Полный маршрут передачи данных складывается из одношаговых решений, принятых компонентами сети.

Маршрутизация от источника передачи данных легко реализуется на промежуточных компонентах сети, но требует полного знания маршрутов на конечных компонентах. Она достаточно редко используется в современных сетевых системах, и далее мы ее рассматривать не будем.

Для работы алгоритмов одношаговой маршрутизации, которые являются основой соответствующих протоколов, на каждом компоненте сети, имеющем возможность передавать информацию более чем одному компоненту, обычно строится специальная таблица маршрутов (см. таблицу 14.1). В простейшем случае каждая запись такой таблицы содержит: адрес вычислительного комплекса получателя; адрес компонента сети, напрямую подсоединенного к данному, которому следует отправить

Таблица 14.1

Адресат назначения	Адрес очередного компонента	Адрес исходящей линии связи
5-12	20	21
1-4	15	22
default	28	24

пакет, предназначенный для этого получателя; указание о том, по какой линии связи (через какой сетевой адаптер) должен быть отправлен пакет. Поскольку получателей в сети существует огромное количество, для сокращения числа записей в таблице маршрутизации обычно прибегают к двум специальным приемам.

Во-первых, числовые адреса топологически близко расположенных комплексов (например, комплексов, принадлежащих одной локальной вычислительной сети) стараются выбирать из последовательного диапазона адресов. В этом случае запись в таблице маршрутизации может содержать не адрес конкретного получателя, а диапазон адресов для некоторой сети (номер сети).

Во-вторых, если для очень многих получателей в качестве очередного узла маршрута используется один и тот же компонент сети, а остальные маршруты выбираются для ограниченного числа получателей, то в таблицу явно заносятся только записи для этого небольшого количества получателей, а для маршрута, ведущего к большей части всей сети, делается одна запись – маршрутизация по умолчанию (default). Пример простой таблицы маршрутизации для некоторого комплекса некой абстрактной сети приведен ниже.

По способам формирования и использования таблиц маршрутизации алгоритмы одношаговой маршрутизации можно разделить на три класса:

- алгоритмы фиксированной маршрутизации;
- алгоритмы простой маршрутизации;
- алгоритмы динамической маршрутизации.

При *фиксированной маршрутизации* таблица, как правило, создается в процессе загрузки операционной системы. Все записи в ней являются статическими. Линия связи, которая будет использоваться для доставки информации от данного узла к некоторому узлу A в сети, выбирается раз и навсегда. Обычно линии выбирают так, чтобы минимизировать полное время доставки данных. Преимуществом этой стратегии является простота реализации. Основной же недостаток заключается в том, что при отка-

зе выбранной линии связи данные не будут доставлены, даже если существует другой физический путь для их передачи.

В алгоритмах *простой маршрутизации* таблица либо не используется совсем, либо строится на основе анализа адресов отправителей приходящих пакетов информации. Различают несколько видов простой маршрутизации – *случайную*, *лавинную* и *маршрутизацию по прецедентам*. При случайной маршрутизации прибывший пакет отсылается в первом попавшемся направлении, кроме исходного. При лавинной маршрутизации один и тот же пакет рассылается по всем направлениям, кроме исходного. Случайная и лавинная маршрутизации, естественно, не используют таблиц маршрутов. При маршрутизации по прецедентам таблица маршрутизации строится по предыдущему опыту, исходя из анализа адресов отправителей приходящих пакетов. Если прибывший пакет адресован компоненту сети, от которого когда-либо приходили данные, то соответствующая запись об этом содержится в таблице маршрутов, и для дальнейшей передачи пакета выбирается линия связи, указанная в таблице. Если такой записи нет, то пакет может быть отослан случайным или лавинным способом. Алгоритмы простой маршрутизации действительно просты в реализации, но отнюдь не гарантируют доставку пакета указанному адресату за приемлемое время и по рациональному маршруту без перегрузки сети.

Наиболее гибкими являются алгоритмы *динамической* или *адаптивной маршрутизации*, которые умеют обновлять содержимое таблиц маршрутов на основе обработки специальных сообщений, приходящих от других компонентов сети, занимающихся маршрутизацией, удовлетворяющих определенному протоколу. Такие алгоритмы принято делить на два подкласса: *алгоритмы дистанционно-векторные* и *алгоритмы состояния связей*.

При дистанционно-векторной маршрутизации компоненты операционных систем на соседних вычислительных комплексах сети, занимающиеся выбором маршрута (их принято называть *маршрутизатор* или *router*), периодически обмениваются векторами, которые представляют собой информацию о расстояниях от данного компонента до всех известных ему адресатов в сети. Под расстоянием обычно понимается количество переходов между компонентами сети (*hops*), которые необходимо сделать, чтобы достичь адресата, хотя возможно существование и других метрик, включающих скорость и/или стоимость передачи пакета по линии связи. Каждый такой вектор формируется на основании таблицы маршрутов. Пришедшие от других комплексов векторы модернизируются с учетом расстояния, которое они прошли при последней передаче. Затем в таблицу маршрутизации вносятся изменения, так чтобы в ней содержались только маршруты с кратчайшими расстояниями. При доста-

точно длительной работе каждый маршрутизатор будет иметь таблицу маршрутизации с оптимальными маршрутами ко всем потенциальным адресатам.

Векторно-дистанционные протоколы обеспечивают достаточно разумную маршрутизацию пакетов, но не способны предотвратить возможность возникновения маршрутных петель при сбоях в работе сети. Поэтому векторно-дистанционная машрутизация может быть эффективна только в относительно небольших сетях. Для больших сетей применяются алгоритмы состояния связей, на каждом маршрутизаторе строящие графы сети, в качестве узлов которого выступают ее компоненты, а в качестве ребер, обладающих стоимостью, существующие между ними линии связи. Маршрутизаторы периодически обмениваются графами и вносят в них изменения. Выбор маршрута связан с поиском оптимального по стоимости пути по такому графу.

Подробное описание протоколов динамической маршрутизации можно найти в [Олифер, 2002] и [Таненбаум, 2003].

Обычно вычислительные сети используют смесь различных стратегий маршрутизации. Для одних адресов назначения может использоваться фиксированная маршрутизация, для других – простая, для третьих – динамическая. В локальных вычислительных сетях обычно используются алгоритмы фиксированной маршрутизации, в отличие от глобальных вычислительных сетей, в которых в основном применяют алгоритмы адаптивной маршрутизации. Протоколы маршрутизации относятся к сетевому уровню эталонной модели.

Связь с установлением логического соединения и передача данных с помощью сообщений

Рассказывая об отличиях взаимодействия локальных и удаленных процессов, мы упомянули, что в основе всех средств связи на автономном компьютере так или иначе лежит механизм совместного использования памяти, в то время как в основе всех средств связи между удаленными процессами лежит передача сообщений. Неудивительно, что количество категорий средств удаленной связи сокращается до одной – канальных средств связи. Обеспечивать интерфейс для сигнальных средств связи и разделяемой памяти, базируясь на передаче пакетов данных, становится слишком сложно и дорого.

Рассматривая канальные средства связи для локальных процессов в лекции 4, мы говорили о существовании двух моделей передачи данных по каналам связи (теперь мы можем говорить о двух принципиально разных видах протоколов организации канальной связи): поток ввода-вывода и сообщения. Для общения удаленных процессов применяются обе

модели, однако теперь уже более простой моделью становится передача информации с помощью сообщений. Реализация различных моделей происходит на основе протоколов транспортного уровня OSI/ISO.

Транспортные протоколы связи удаленных процессов, которые предназначены для обмена сообщениями, получили наименование *протоколов без установления логического соединения* (*connectionless*) или *протоколов обмена датаграммами*, поскольку само сообщение здесь принято называть *датаграммой* (*datagramm*) или *дейтаграммой*. Каждое сообщение адресуется и посылается процессом индивидуально. С точки зрения операционных систем все датаграммы – это независимые единицы, не имеющие ничего общего с другими датаграммами, которыми обмениваются эти же процессы.

Необходимо отметить, что с точки зрения процессов, обменивающихся информацией, датаграммы, конечно, могут быть связаны по содержанию друг с другом, но ответственность за установление и поддержание этой семантической связи лежит не на сетевых частях операционных систем, а на самих пользовательских взаимодействующих процессах (вышестоящие уровни эталонной модели).

По-другому обстоит дело с транспортными протоколами, которые поддерживают потоковую модель. Они получили наименование *протоколов, требующих установления логического соединения* (*connection-oriented*). В их основе лежит передача данных с помощью пакетов информации. Но операционные системы сами нарезают эти пакеты из передаваемого потока данных, организовывают правильную последовательность их получения и снова объединяют полученные пакеты в поток, так что с точки зрения взаимодействующих процессов после установления логического соединения они имеют дело с потоковым средством связи, напоминающим pipe или FIFO. Эти протоколы должны обеспечивать надежную связь.

Синхронизация удаленных процессов

Мы рассмотрели основные принципы логической организации сетевых средств связи, внешние по отношению к взаимодействующим процессам. Однако, как отмечалось в лекции 5, для корректной работы таких процессов необходимо обеспечить определенную их синхронизацию, которая устранила бы возникновение race condition на соответствующих критических участках. Вопросы синхронизации удаленных процессов обычно рассматриваются в курсах, посвященных распределенным операционным системам. Интересующиеся этими вопросами могут обратиться к книгам [Silberschatz, 2002] и [Таненбаум II, 2003].

Заключение

Основными причинами объединения компьютеров в вычислительные сети являются потребности в разделении ресурсов, ускорении вычислений, повышении надежности и облегчении общения пользователей.

Вычислительные комплексы в сети могут находиться под управлением сетевых или распределенных вычислительных систем. Основой для объединения компьютеров в сеть служит взаимодействие удаленных процессов. При рассмотрении вопросов организации взаимодействия удаленных процессов нужно принимать во внимание основные отличия их кооперации от кооперации локальных процессов.

Базой для взаимодействия локальных процессов служит организация общей памяти, в то время как для удаленных процессов – это обмен физическими пакетами данных.

Организация взаимодействия удаленных процессов требует от сетевых частей операционных систем поддержки определенных протоколов. Сетевые средства связи обычно строятся по «слоеному» принципу. Формальный перечень правил, определяющих последовательность и формат сообщений, которыми обмениваются сетевые компоненты различных вычислительных систем, лежащие на одном уровне, называется сетевым протоколом. Каждый уровень слоеной системы может взаимодействовать непосредственно только со своими вертикальными соседями, руководствуясь четко закрепленными соглашениями – вертикальными протоколами или интерфейсами. Вся совокупность интерфейсов и сетевых протоколов в сетевых системах, построенных по слоеному принципу, достаточная для организации взаимодействия удаленных процессов, образует *семейство протоколов* или *стек протоколов*.

Удаленные процессы, в отличие от локальных, при взаимодействии обычно требуют двухуровневой адресации при своем общении. Полный адрес процесса состоит из двух частей: удаленной и локальной.

Для удаленной адресации используются символьные и числовые имена узлов сети. Перевод имен из одной формы в другую (разрешение имен) может осуществляться с помощью централизованно обновляемых таблиц соответствия полностью на каждом узле или с использованием выделения зон ответственности специальных серверов. Для локальной адресации процессов применяются порты. Упорядоченная пара из адреса узла в сети и порта получила название socket.

Для доставки сообщения от одного узла к другому могут использоваться различные протоколы маршрутизации.

С точки зрения пользовательских процессов обмен информацией может осуществляться в виде датаграмм или потока данных.

Часть VII. Проблемы безопасности операционных систем

По-настоящему безопасной можно считать лишь систему, которая выключена, замурована в бетонный корпус, заперта в помещении со свинцовыми стенами и охраняется вооруженным караулом... Но и в этом случае сомнения не оставляют меня.

Юджин Х. Спаффорд

Лекция 15. Основные понятия информационной безопасности

Рассмотрены подходы к обеспечению безопасности информационных систем. Ключевые понятия информационной безопасности: конфиденциальность, целостность и доступность информации, а любое действие, направленное на их нарушение, называется угрозой. Основные понятия информационной безопасности регламентированы в основополагающих документах. Существует несколько базовых технологий безопасности, среди которых можно выделить криптографию.

Ключевые слова: информационная безопасность, угроза, атака, целостность, конфиденциальность, доступность, оранжевая книга, классы безопасности, C2, криптография, симметричный ключ, RSA, односторонняя функция.

Введение

В октябре 1988 года в США произошло событие, названное специалистами крупнейшим нарушением безопасности американских компьютерных систем из когда-либо случавшихся. 23-летний студент выпускного курса Корнельского университета Роберт Т. Моррис запустил в компьютерной сети ARPANET программу, представлявшую собой редко встречающуюся разновидность компьютерных вирусов – сетевых «червей». В результате атаки был полностью или частично заблокирован ряд общенациональных компьютерных сетей, в частности Internet, CSnet, NSFnet, BITnet, ARPANET и несекретная военная сеть Milnet. В итоге вирус поразил более 6200 компьютерных систем по всей Америке, включая системы многих крупнейших университетов, институтов, правительственных

лабораторий, частных фирм, военных баз, клиник, агентства NASA. Общий ущерб от этой атаки оценивается специалистами минимум в 100 млн. долларов Р. Моррис был исключен из университета с правом повторного поступления через год и приговорен судом к штрафу в 270 тыс. долларов и трем месяцам тюремного заключения.

Важность решения проблемы информационной безопасности в настоящее время общепризнана, подтверждением чему служат громкие процессы о нарушении целостности систем. Убытки ведущих компаний в связи с нарушениями безопасности информации составляют триллионы долларов, причем только треть опрошенных компаний смогли определить количественно размер потерь. Проблема обеспечения безопасности носит комплексный характер, для ее решения необходимо сочетание законодательных, организационных и программно-технических мер.

Таким образом, обеспечение информационной безопасности требует системного подхода и нужно использовать разные средства и приемы — морально-этические, законодательные, административные и технические. Нас будут интересовать последние. Технические средства реализуются программным и аппаратным обеспечением и решают разные задачи по защите, они могут быть встроены в операционные системы либо могут быть реализованы в виде отдельных продуктов. Во многих случаях центр тяжести смещается в сторону защищенности операционных систем.

Есть несколько причин для реализации дополнительных средств защиты. Наиболее очевидная — помешать внешним попыткам нарушить доступ к конфиденциальной информации. Не менее важно, однако, гарантировать, что каждый программный компонент в системе использует системные ресурсы только способом, совместимым с установленной политикой применения этих ресурсов. Такие требования абсолютно необходимы для надежной системы. Кроме того, наличие защитных механизмов может увеличить надежность системы в целом за счет обнаружения скрытых ошибок интерфейса между компонентами системы. Раннее обнаружение ошибок может предотвратить «заражение» неисправной подсистемой остальных.

Политика в отношении ресурсов может меняться в зависимости от приложения и с течением времени. Операционная система должна обеспечивать прикладные программы инструментами для создания и поддержки защищенных ресурсов. Здесь реализуется важный для гибкости системы принцип — отделение политики от механизмов. Механизмы определяют, как может быть сделано что-либо, тогда как политика решает, что должно быть сделано. Политика может меняться в зависимости от места и времени. Желательно, чтобы были реализованы по возможности общие механизмы, тогда как изменение политики требует лишь модификации системных параметров или таблиц.

К сожалению, построение защищенной системы предполагает необходимость склонить пользователя к отказу от некоторых интересных возможностей. Например, письмо, содержащее в качестве приложения документ в формате Word, может включать макросы. Открытие такого письма влечет за собой запуск чужой программы, что потенциально опасно. То же самое можно сказать про web-страницы, содержащие апплеты. Вместо критического отношения к использованию такой функциональности пользователи современных компьютеров предпочитают периодически запускать антивирусные программы и читать успокаивающие статьи о безопасности Java.

Угрозы безопасности

Знание возможных угроз, а также уязвимых мест защиты, которые эти угрозы обычно эксплуатируют, необходимо для того, чтобы выбирать наиболее экономичные средства обеспечения безопасности.

Считается, что безопасная система должна обладать свойствами *конфиденциальности*, *доступности* и *целостности*. Любое потенциальное действие, которое направлено на нарушение конфиденциальности, целостности и доступности информации, называется угрозой. Реализованная угроза называется атакой.

Конфиденциальная (*confidentiality*) система обеспечивает уверенность в том, что секретные данные будут доступны только тем пользователям, которым этот доступ разрешен (такие пользователи называются *авторизованными*). Под *доступностью* (*availability*) понимают гарантию того, что авторизованным пользователям всегда будет доступна информация, которая им необходима. И наконец, *целостность* (*integrity*) системы подразумевает, что неавторизованные пользователи не могут каким-либо образом модифицировать данные.

Защита информации ориентирована на борьбу с так называемыми умышленными угрозами, то есть с теми, которые, в отличие от случайных угроз (ошибок пользователя, сбоев оборудования и др.), преследуют цель нанести ущерб пользователям ОС.

Умышленные угрозы подразделяются на активные и пассивные. *Пассивная* угроза – несанкционированный доступ к информации без изменения состояния системы, *активная* – несанкционированное изменение системы. Пассивные атаки труднее выявить, так как они не влекут за собой никаких изменений данных. Защита против пассивных атак базируется на средствах их предотвращения.

Можно выделить несколько типов угроз. Наиболее распространенная угроза – *попытка проникновения в систему под видом легального пользователя*, например попытки угадывания и подбора паролей. Более сложный вариант – внедрение в систему программы, которая выводит на экран

слово login. Многие легальные пользователи при этом начинают пытаться входить в систему, и их попытки могут протоколироваться. Такие безобидные с виду программы, выполняющие нежелательные функции, называются «троянскими конями». Иногда удается торпедировать работу программы проверки пароля путем многократного нажатия клавиш del, break, cancel и т. д. Для защиты от подобных атак ОС запускает процесс, называемый *аутентификацией* пользователя (см. лекцию 16, раздел «Идентификация и аутентификация»).

Угрозы другого рода связаны с *нежелательными действиями легальных пользователей*, которые могут, например, предпринимать попытки чтения страниц памяти, дисков и лент, которые сохранили информацию, связанную с предыдущим использованием. Защита в таких случаях базируется на надежной системе *авторизации* (см. лекцию 16, раздел «Авторизация. Разграничение доступа к объектам ОС»). В эту категорию также попадают атаки типа ***отказ в обслуживании***, когда сервер затоплен мощным потоком запросов и становится фактически недоступным для отдельных авторизованных пользователей.

Наконец, функционирование системы может быть нарушено с помощью *программ-вирусов* или *программ-червей*, которые специально предназначены для того, чтобы причинить вред или недолжным образом использовать ресурсы компьютера. Общее название угроз такого рода – вредоносные программы (malicious software). Обычно они распространяются сами по себе, переходя на другие компьютеры через зараженные файлы, дискеты или по электронной почте. Наиболее эффективный способ борьбы с подобными программами – соблюдение правил «компьютерной гигиены». Многопользовательские компьютеры меньше страдают от вирусов по сравнению с персональными, поскольку там имеются системные средства защиты.

Таковы основные угрозы, на долю которых приходится львиная доля ущерба, наносимого информационным системам.

В ряде монографий [Столлингс, 2002; Таненбаум, 2002] также рассматривается модель злоумышленника, поскольку очевидно, что необходимые для организации защиты усилия зависят от предполагаемого противника. Обеспечение безопасности имеет и правовые аспекты, но это уже выходит за рамки данного курса.

Формализация подхода к обеспечению информационной безопасности

Проблема информационной безопасности оказалась настолько важной, что в ряде стран были выпущены основополагающие документы, в которых регламентированы основные подходы к проблеме информаци-

онной безопасности. В результате оказалось возможным ранжировать информационные системы по степени надежности.

Наиболее известна оранжевая (по цвету обложки) книга Министерства обороны США [DoD, 1993]. В этом документе определяется четыре уровня безопасности – D, C, B и A. По мере перехода от уровня D до A к надежности систем предъявляются все более жесткие требования. Уровни C и B подразделяются на классы (C1, C2, B1, B2, B3). Чтобы система в результате процедуры сертификации могла быть отнесена к некоторому классу, ее защита должна удовлетворять оговоренным требованиям.

В качестве примера рассмотрим требования класса C2, которому удовлетворяют ОС Windows NT, отдельные реализации Unix и ряд других.

- Каждый пользователь должен быть идентифицирован уникальным входным именем и паролем для входа в систему. Доступ к компьютеру предоставляется лишь после аутентификации.
- Система должна быть в состоянии использовать эти уникальные идентификаторы, чтобы следить за действиями пользователя (управление избирательным доступом). Владелец ресурса (например, файла) должен иметь возможность контролировать доступ к этому ресурсу.
- Операционная система должна защищать объекты от повторного использования. Перед выделением новому пользователю все объекты, включая память и файлы, должны инициализироваться.
- Системный администратор должен иметь возможность вести учет всех событий, относящихся к безопасности.
- Система должна защищать себя от внешнего влияния или навязывания, такого как модификация загруженной системы или системных файлов, хранящихся на диске.

Сегодня на смену оранжевой книге пришел стандарт Common Criteria, а набор критериев Controlled Access Protection Profile сменил критерии класса C2.

Основополагающие документы содержат определения многих ключевых понятий, связанных с информационной безопасностью. Некоторые из них (аутентификация, авторизация, домен безопасности и др.) будут рассмотрены в следующей лекции. В дальнейшем мы также будем оперировать понятиями «субъект» и «объект» безопасности. Субъект безопасности – активная системная составляющая, к которой применяется политика безопасности, а объект – пассивная. Примерами субъектов могут служить пользователи и группы пользователей, а объектов – файлы, системные таблицы, принтер и т. п.

По существу, проектирование системы безопасности подразумевает ответы на следующие вопросы: какую информацию защищать, какого рода атаки на безопасность системы могут быть предприняты, какие средства использовать для защиты каждого вида информации? Поиск ответов

на данные вопросы называется формированием *политики безопасности*, которая помимо чисто технических аспектов включает также и решение организационных проблем. На практике реализация политики безопасности состоит в присвоении субъектам и объектам идентификаторов и фиксации набора правил, позволяющих определить, имеет ли данный субъект авторизацию, достаточную для предоставления к данному объекту указанного типа доступа.

Формируя политику безопасности, необходимо учитывать несколько базовых принципов. Так, Зальтцер (Saltzer) и Шредер (Schroeder) (1975) на основе своего опыта работы с MULTICS сформулировали следующие рекомендации для проектирования системы безопасности ОС:

- Проектирование системы должно быть открытым. Нарушитель и так все знает (криптографические алгоритмы открыты).
- Не должно быть доступа по умолчанию. Ошибки с отклонением легитимного доступа будут обнаружены скорее, чем ошибки там, где разрешен неавторизованный доступ.
- Нужно тщательно проверять текущее авторство. Так, многие системы проверяют привилегии доступа при открытии файла и не делают этого после. В результате пользователь может открыть файл и держать его открытым в течение недели и иметь к нему доступ, хотя владелец уже сменил защиту.
- Давать каждому процессу минимум возможных привилегий.
- Защитные механизмы должны быть просты, постоянны и встроены в нижний слой системы, это не аддитивные добавки (известно много неудачных попыток «улучшения» защиты слабо приспособленной для этого ОС MS-DOS).
- Важна физиологическая приемлемость. Если пользователь видит, что защита требует слишком больших усилий, он от нее откажется. Ущерб от атаки и затраты на ее предотвращение должны быть сбалансированы.

Приведенные соображения показывают необходимость продумывания и встраивания защитных механизмов на самых ранних стадиях проектирования системы.

Криптография как одна из базовых технологий безопасности ОС

Многие службы информационной безопасности, такие как контроль входа в систему, разграничение доступа к ресурсам, обеспечение безопасного хранения данных и ряд других, опираются на использование криптографических алгоритмов. Имеется обширная литература по этому актуальному для безопасности информационных систем вопросу.

Шифрование – процесс преобразования сообщения из открытого текста (plaintext) в шифротекст (ciphertext) таким образом, чтобы:

- его могли прочитать только те стороны, для которых оно предназначено;
- проверить подлинность отправителя (аутентификация);
- гарантировать, что отправитель действительно послал данное сообщение.

В алгоритмах шифрования предусматривается наличие *ключа*. Ключ – это некий параметр, не зависящий от открытого текста. Результат применения алгоритма шифрования зависит от используемого ключа. В криптографии принято правило Кирхгофа: «Стойкость шифра должна определяться только секретностью ключа». Правило Кирхгофа подразумевает, что алгоритмы шифрования должны быть открыты.

В методе шифрования с *секретным* или *симметричным* ключом имеется один ключ, который используется как для шифрования, так и для расшифровки сообщения. Такой ключ нужно хранить в секрете. Это затрудняет использование системы шифрования, поскольку ключи должны регулярно меняться, для чего требуется их секретное распространение. Наиболее популярные алгоритмы шифрования с секретным ключом: DES, TripleDES, ГОСТ и ряд других.

Часто используется шифрование с помощью *односторонней* функции, называемой также хеш- или дайджест-функцией. Применение этой функции к шифруемым данным позволяет сформировать небольшой дайджест из нескольких байтов, по которому невозможно восстановить исходный текст. Получатель сообщения может проверить целостность данных, сравнивая полученный вместе с сообщением дайджест с вычисленным вновь при помощи той же односторонней функции. Эта техника активно используется для контроля входа в систему. Например, пароли пользователей хранятся на диске в зашифрованном односторонней функцией виде. Наиболее популярные хеш-функции: MD4, MD5 и др.

В системах шифрования с *открытым* или *асимметричным* ключом (public/assymmetric key) используется два ключа (см. рис. 15.1). Один из ключей, называемый открытым, несекретным, используется для шифрования сообщений, которые могут быть расшифрованы только с помощью секретного ключа, имеющегося у получателя, для которого предназначено сообщение. Иногда поступают по-другому. Для шифрования сообщения используется секретный ключ, и если сообщение можно расшифровать с помощью открытого ключа, подлинность отправителя будет гарантирована (система электронной подписи). Этот принцип изобретен Уитфилдом Диффи (Whitfield Diffie) и Мартином Хеллманом (Martin Hellman) в 1976 году.

Использование открытых ключей снимает проблему обмена и хранения ключей, свойственную системам с симметричными ключами. Откры-

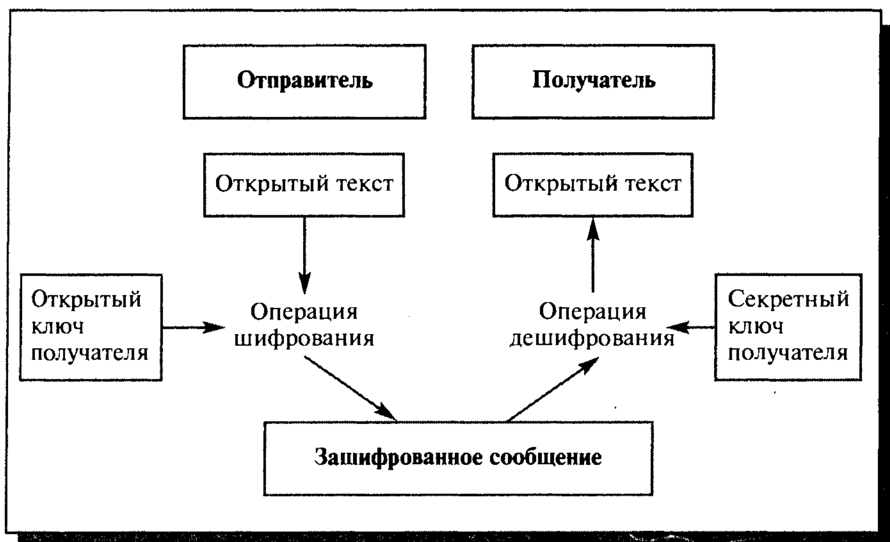


Рис. 15.1. Шифрование открытым ключом

тые ключи могут храниться публично, и каждый может послать зашифрованное открытым ключом сообщение владельцу ключа. Однако расшифровать это сообщение может только владелец открытого ключа при помощи своего секретного ключа, и никто другой. Несмотря на очевидные удобства, связанные с хранением и распространением ключей, асимметричные алгоритмы гораздо менее эффективны, чем симметричные, поэтому во многих криптографических системах используются оба метода.

Среди несимметричных алгоритмов наиболее известен RSA, предложенный Роном Ривестом (Ron Rivest), Ади Шамиром (Adi Shamir) и Леонардом Эдлманом (Leonard Adleman). Рассмотрим его более подробно.

Шифрование с использованием алгоритма RSA

Идея, положенная в основу метода, состоит в том, чтобы найти такую функцию $y = f(x)$, для которой получение обратной функции $x = f^{-1}(y)$ было бы в общем случае очень сложной задачей (NP-полной задачей). Например, получить произведение двух чисел $n = p \times q$ просто, а разложить n на множители, если p и q достаточно большие простые числа, — NP-полная задача с вычислительной сложностью ~ n^{10}. Однако если знать некую секретную информацию, то найти обратную функцию $x = f^{-1}(y)$ существенно проще. Такие функции также называют односторонними функциями с лазейкой или потайным ходом.

Применяемые в RSA прямая и обратная функции просты. Они базируются на применении теоремы Эйлера из теории чисел.

Прежде чем сформулировать теорему Эйлера, необходимо определить важную функцию $\phi(n)$ из теории чисел, называемую функцией Эйлера. Это число взаимно простых (взаимно простыми называются целые числа, не имеющие общих делителей) с n целых чисел, меньших n. Например, $\phi(7) = 6$. Очевидно, что, если p и q – простые числа и $p \neq q$, то $\phi(p) = p-1$, и $\phi(pq)=(p-1)\times(q-1)$.

Теорема Эйлера

Теорема Эйлера утверждает, что для любых взаимно простых чисел x и n $(x < n)$

$$x^{\phi(n)} \bmod n = 1$$

или в более общем виде

$$x^{k\phi(n)+1} \bmod n = 1.$$

Сформулируем еще один важный результат. Для любого $m > 0$ и $0 < e < m$, где e и m взаимно просты, найдется единственное $0 < d < m$, такое, что

$$de \bmod m = 1.$$

Здесь d легко можно найти по обобщенному алгоритму Евклида (см., например, Д. Кнут. Искусство программирования на ЭВМ, т. 2, 4.5.2). Известно, что вычислительная сложность алгоритма Евклида ~ ln n.

Подставляя $\phi(n)$ вместо m, получим

$$de \bmod \phi(n)=1$$

или

$$de = k\phi(n)+1.$$

Тогда прямой функцией будет

$$f(x) = x^e \bmod n$$

где x – положительное целое, $x < n = pq$, p и q – целые простые числа и, следовательно,

$\phi(n)=(p-1)(q-1)$

где e – положительное целое и $e < \phi(n)$. Здесь e и n открыты. Однако p и q неизвестны (чтобы их найти, нужно выполнить разбиение n на множители), следовательно, неизвестна и $\phi(n)$, а именно они и составляют потайной ход.

Вычислим обратную функцию

$$f^{-1}(y) = y^d \bmod n = x^{ed} \bmod n = x^{k\phi(n)+1} \bmod n = x.$$

Последнее преобразование справедливо, поскольку $x < n$ и x и n взаимно просты.

При практическом использовании алгоритма RSA вначале необходимо выполнить генерацию ключей. Для этого нужно:

1) Выбрать два очень больших простых числа p и q;
2) Вычислить произведение $n = p \times q$;
3) Выбрать большое случайное число d, не имеющее общих сомножителей с числом $(p-1)\times(q-1)$;
4) Определить число e, чтобы выполнялось $(e\times d) \bmod ((p-1)\times(q-1))=1$.

Тогда открытым ключом будут числа e и n, а секретным ключом – числа d и n.

Теперь, чтобы зашифровать данные по известному ключу $\{e,n\}$, необходимо сделать следующее:

- Разбить шифруемый текст на блоки, где i-й блок представить в виде числа M, величина которого меньше, чем n. Это можно сделать различными способами, например используя вместо букв их номера в алфавите.
- Зашифровать текст, рассматриваемый как последовательность чисел $m(i)$, по формуле $c(i)=(m(i)^e) \bmod n$.

Чтобы расшифровать эти данные, используя секретный ключ $\{d,n\}$, необходимо вычислить: $m(i) = (c(i)^d) \bmod n$. В результате будет получено множество чисел $m(i)$, которые представляют собой часть исходного текста.

Например, зашифруем и расшифруем сообщение «АБВ», которое представим как число 123.

Выбираем $p = 5$ и $q = 11$ (числа на самом деле должны быть большими).

Находим $n = 5\times11 = 55$.

Определяем $(p-1)\times(q-1) = 40$. Тогда d будет равно, например, 7.

Выберем e, исходя из $(e\times7) \bmod 40 = 1$. Например, $e = 3$.

Теперь зашифруем сообщение, используя открытый ключ $\{3,55\}$:

$$C_1 = (1^3) \bmod 55 = 1$$

$C_2 = (2^3) \bmod 55 = 8$
$C_3 = (3^3) \bmod 55 = 27.$

Теперь расшифруем эти данные, используя закрытый ключ {7,55}:

$M_1 = (1^7) \bmod 55 = 1$
$M_2 = (8^7) \bmod 55 = 2097152 \bmod 55 = 2$
$M_3 = (27^7) \bmod 55 = 10460353203 \bmod 55 = 3.$

Таким образом, все данные расшифрованы.

Заключение

Информационная безопасность относится к числу дисциплин, развивающихся чрезвычайно быстрыми темпами. Только комплексный, систематический, современный подход способен успешно противостоять нарастающим угрозам.

Ключевые понятия информационной безопасности: конфиденциальность, целостность и доступность информации, а любое действие, направленное на их нарушение, называется угрозой.

Основные понятия информационной безопасности регламентированы в основополагающих документах.

Существует несколько базовых технологий безопасности, среди которых можно выделить криптографию.

Лекция 16. Защитные механизмы операционных систем

Решение вопросов безопасности операционных систем обусловлено их архитектурными особенностями и связано с правильной организацией идентификации и аутентификации, авторизации и аудита.

Ключевые слова: идентификация, аутентификация, аудит, уязвимость паролей, дискреционный доступ, полномочный доступ, домен безопасности, матрица управления доступом, мандаты, перечни возможностей, повторное использование объектов.

Перейдем к описанию системы защиты операционных систем. Ее основными задачами являются идентификация, аутентификация, разграничение доступа пользователей к ресурсам, протоколирование и аудит самой системы. Более подробную информацию об этом можно найти в [Дейтел, 1987], [Столлингс, 2002], [Таненбаум, 2002] и ряде других источников.

Идентификация и аутентификация

Для начала рассмотрим проблему контроля доступа в систему. Наиболее распространенным способом контроля доступа является процедура регистрации. Обычно каждый пользователь в системе имеет уникальный идентификатор. Идентфикаторы пользователей применяются с той же целью, что и идентификаторы любых других объектов, файлов, процессов. *Идентификация* заключается в сообщении пользователем своего идентификатора. Для того чтобы установить, что пользователь именно тот, за кого себя выдает, то есть что именно ему принадлежит введенный идентификатор, в информационных системах предусмотрена процедура *аутентификации* (authentication, опознавание, в переводе с латинского означает «установление подлинности»), задача которой – предотвращение доступа к системе нежелательных лиц.

Обычно аутентификация базируется на одном или более из трех пунктов:

- то, чем пользователь владеет (ключ или магнитная карта);
- то, что пользователь знает (пароль);
- атрибуты пользователя (отпечатки пальцев, подпись, голос).

Пароли, уязвимость паролей

Наиболее простой подход к аутентификации – применение пользовательского пароля.

Когда пользователь идентифицирует себя при помощи уникального идентификатора или имени, у него запрашивается пароль. Если пароль, сообщенный пользователем, совпадает с паролем, хранящимся в системе, система предполагает, что пользователь легитимен. Пароли часто используются для защиты объектов в компьютерной системе в отсутствие более сложных схем защиты.

Недостатки паролей связаны с тем, что трудно сохранить баланс между удобством пароля для пользователя и его надежностью. Пароли могут быть угаданы, случайно показаны или нелегально переданы авторизованным пользователем неавторизованному.

Есть два общих способа угадать пароль. Один связан со сбором информации о пользователе. Люди обычно используют в качестве паролей очевидную информацию (скажем, имена животных или номерные знаки автомобилей). Для иллюстрации важности разумной политики назначения идентификаторов и паролей можно привести данные исследований, проведенных в AT&T, показывающие, что из 500 попыток несанкционированного доступа около 300 составляют попытки угадывания паролей или беспарольного входа по пользовательским именам guest, demo и т. д.

Другой способ – попытаться перебрать все наиболее вероятные комбинации букв, чисел и знаков пунктуации (атака по словарю). Например, четыре десятичные цифры дают только 10 000 вариантов, более длинные пароли, введенные с учетом регистра символов и пунктуации, не столь уязвимы, но тем не менее таким способом удается разгадать до 25% паролей. Чтобы заставить пользователя выбрать трудноугадываемый пароль, во многих системах внедрена реактивная проверка паролей, которая при помощи собственной программы-взломщика паролей может оценить качество пароля, введенного пользователем.

Несмотря на все это, пароли распространены, поскольку они удобны и легко реализуемы.

Шифрование пароля

Для хранения секретного списка паролей на диске во многих ОС используется криптография. Система задействует одностороннюю функцию, которую просто вычислить, но для которой чрезвычайно трудно (разработчики надеются, что невозможно) подобрать обратную функцию.

Например, в ряде версий Unix в качестве односторонней функции используется модифицированный вариант алгоритма DES. Введенный пароль

длиной до 8 знаков преобразуется в 56-битовое значение, которое служит входным параметром для процедуры crypt(), основанной на этом алгоритме. Результат шифрования зависит не только от введенного пароля, но и от случайной последовательности битов, называемой привязкой (переменная salt). Это сделано для того, чтобы решить проблему совпадающих паролей. Очевидно, что саму привязку после шифрования необходимо сохранять, иначе процесс не удастся повторить. Модифицированный алгоритм DES выполняется, имея входное значение в виде 64-битового блока нулей, с использованием пароля в качестве ключа, а на каждой следующей итерации входным параметром служит результат предыдущей итерации. Всего процедура повторяется 25 раз. Полученное 64-битовое значение преобразуется в 11 символов и хранится рядом с открытой переменной salt.

В ОС Windows NT преобразование исходного пароля также осуществляется многократным применением алгоритма DES и алгоритма MD4.

Хранятся только кодированные пароли. В процессе аутентификации представленный пользователем пароль кодируется и сравнивается с хранящимися на диске. Таким образом, файл паролей нет необходимости держать в секрете.

При удаленном доступе к ОС нежелательна передача пароля по сети в открытом виде. Одним из типовых решений является использование криптографических протоколов. В качестве примера можно рассмотреть протокол опознавания с подтверждением установления связи путем вызова – CHAP (Challenge Handshake Authenfication Protocol).

Опознавание достигается за счет проверки того, что у пользователя, осуществляющего доступ к серверу, имеется секретный пароль, который уже известен серверу.

Пользователь инициирует диалог, передавая серверу свой идентификатор. В ответ сервер посылает пользователю запрос (вызов), состоящий из идентифицирующего кода, случайного числа и имени узла сервера или имени пользователя. При этом пользовательское оборудование в результате запроса пароля пользователя отвечает следующим ответом, зашифрованным с помощью алгоритма одностороннего хеширования, наиболее распространенным видом которого является MD5. После получения ответа сервер при помощи той же функции с теми же аргументами шифрует собственную версию пароля пользователя. В случае совпадения результатов вход в систему разрешается. Существенно, что незашифрованный пароль при этом по каналу связи не посылается.

В микротелефонных трубках используется аналогичный метод.

В системах, работающих с большим количеством пользователей, когда хранение всех паролей затруднительно, применяются для опознавания сертификаты, выданные доверенной стороной (см., например, [Столлингс, 2002]).

Авторизация. Разграничение доступа к объектам ОС

После успешной регистрации система должна осуществлять авторизацию (authorization) – предоставление субъекту прав на доступ к объекту. Средства авторизации контролируют доступ легальных пользователей к ресурсам системы, предоставляя каждому из них именно те права, которые были определены администратором, а также осуществляют контроль возможности выполнения пользователем различных системных функций. Система контроля базируется на общей модели, называемой матрицей доступа. Рассмотрим ее более подробно.

Как уже говорилось в предыдущей лекции, компьютерная система может быть смоделирована как набор субъектов (процессы, пользователи) и объектов. Под объектами мы понимаем как ресурсы оборудования (процессор, сегменты памяти, принтер, диски и ленты), так и программные ресурсы (файлы, программы, семафоры), то есть все то, доступ к чему контролируется. Каждый объект имеет уникальное имя, отличающее его от других объектов в системе, и каждый из них может быть доступен через хорошо определенные и значимые операции.

Операции зависят от объектов. Например, процессор может только выполнять команды, сегменты памяти могут быть записаны и прочитаны, считыватель магнитных карт может только читать, а файлы данных могут быть записаны, прочитаны, переименованы и т. д.

Желательно добиться того, чтобы процесс осуществлял авторизованный доступ только к тем ресурсам, которые ему нужны для выполнения его задачи. Это требование минимума привилегий, уже упомянутое в предыдущей лекции, полезно с точки зрения ограничения количества повреждений, которые процесс может нанести системе. Например, когда процесс P вызывает процедуру A, ей должен быть разрешен доступ только к переменным и формальным параметрам, переданным ей, она не должна иметь возможность влиять на другие переменные процесса. Аналогично компилятор не должен оказывать влияния на произвольные файлы, а только на их хорошо определенное подмножество (исходные файлы, листинги и др.), имеющее отношение к компиляции. С другой стороны, компилятор может иметь личные файлы, используемые для оптимизационных целей, к которым процесс P не имеет доступа.

Различают *дискреционный* (избирательный) способ управления доступом и *полномочный* (мандатный).

При дискреционном доступе, подробно рассмотренном ниже, определенные операции над конкретным ресурсом запрещаются или разрешаются субъектам или группам субъектов. С концептуальной точки зрения

текущее состояние прав доступа при дискреционном управлении описывается матрицей, в строках которой перечислены субъекты, в столбцах – объекты, а в ячейках – операции, которые субъект может выполнить над объектом.

Полномочный подход заключается в том, что все объекты могут иметь уровни секретности, а все субъекты делятся на группы, образующие иерархию в соответствии с уровнем допуска к информации. Иногда это называют моделью многоуровневой безопасности, которая должна обеспечивать выполнение следующих правил:

- Простое свойство секретности. Субъект может читать информацию только из объекта, уровень секретности которого не выше уровня секретности субъекта. Генерал читает документы лейтенанта, но не наоборот.
- *-свойство. Субъект может записывать информацию в объекты только своего уровня или более высоких уровней секретности. Генерал не может случайно разгласить нижним чинам секретную информацию.

Некоторые авторы утверждают [Таненбаум, 2002], что последнее требование называют *-свойством, потому что в оригинальном докладе не смогли придумать для него подходящего названия. В итоге во все последующие документы и монографии оно вошло как *-свойство.

Отметим, что данная модель разработана для хранения секретов, но не гарантирует целостности данных. Например, здесь лейтенант имеет право писать в файлы генерала. Более подробно о реализации подобных формальных моделей рассказано в [Столлингс, 2002] и [Таненбаум, 2002].

Большинство операционных систем реализуют именно дискреционное управление доступом. Главное его достоинство – гибкость, основные недостатки – рассредоточенность управления и сложность централизованного контроля.

Домены безопасности

Чтобы рассмотреть схему дискреционного доступа более детально, введем концепцию домена безопасности (protection domain). Каждый домен определяет набор объектов и типов операций, которые могут производиться над каждым объектом. Возможность выполнять операции над объектом есть права доступа, каждое из которых есть упорядоченная пара `<object-name, rights-set>`. Домен, таким образом, есть набор прав доступа. Например, если домен `D` имеет права доступа `<file F, {read, write}>`, это означает, что процесс, выполняемый в домене `D`, может читать или писать в файл `F`, но не может выполнять других операций над этим объектом. Пример доменов можно увидеть на рис. 16.1.

Объект / Домен	F1	F2	F3	Printer
D1	read			
D2				print
D3		read	execute	
D4	read write		read write	

Рис. 16.1. Специфицирование прав доступа к ресурсам

Связь конкретных субъектов, функционирующих в операционных системах, может быть организована следующим образом:

- Каждый пользователь может быть доменом. В этом случае набор объектов, к которым может быть организован доступ, зависит от идентификации пользователя.
- Каждый процесс может быть доменом. В этом случае набор доступных объектов определяется идентификацией процесса.
- Каждая процедура может быть доменом. В этом случае набор доступных объектов соответствует локальным переменным, определенным внутри процедуры. Заметим, что когда процедура выполнена, происходит смена домена.

Рассмотрим стандартную двухрежимную модель выполнения ОС. Когда процесс выполняется в режиме системы (kernel mode), он может выполнять привилегированные инструкции и иметь полный контроль над компьютерной системой. С другой стороны, если процесс выполняется в пользовательском режиме, он может вызывать только непривилегированные инструкции. Следовательно, он может выполняться только внутри предопределенного пространства памяти. Наличие этих двух режимов позволяет защитить ОС (kernel domain) от пользовательских процессов (выполняющихся в user domain). В мультипрограммных системах двух доменов недостаточно, так как появляется необходимость защиты пользователей друг от друга. Поэтому требуется более тщательно разработанная схема.

В ОС Unix домен связан с пользователем. Каждый пользователь обычно работает со своим набором объектов.

Матрица доступа

Модель безопасности, специфицированная в предыдущем разделе (см. рис. 16.1), имеет вид матрицы, которая называется *матрицей доступа*. Какова может быть эффективная реализация матрицы доступа? В общем случае она будет разреженной, то есть большинство ее клеток будут пустыми. Хотя существуют структуры данных для представления разреженной матрицы, они не слишком полезны для приложений, использующих возможности защиты. Поэтому на практике матрица доступа применяется редко. Эту матрицу можно разложить по столбцам, в результате чего получаются *списки прав доступа* (Access Control List – ACL). В результате разложения по строкам получаются *мандаты возможностей* (capability list или capability tickets).

Список прав доступа. Access control list

Каждая колонка в матрице может быть реализована как список доступа для одного объекта. Очевидно, что пустые клетки могут не учитываться. В результате для каждого объекта имеем список упорядоченных пар `<domain, rights-set>`, который определяет все домены с непустыми наборами прав для данного объекта.

Элементами списка могут быть процессы, пользователи или группы пользователей. При реализации широко применяется предоставление доступа по умолчанию для пользователей, права которых не указаны. Например, в Unix все субъекты-пользователи разделены на три группы (владелец, группа и остальные), и для членов каждой группы контролируются операции чтения, записи и исполнения (rwx). В итоге имеем ACL – 9-битный код, который является атрибутом разнообразных объектов Unix.

Мандаты возможностей. Capability list

Как отмечалось выше, если матрицу доступа хранить по строкам, то есть если каждый субъект хранит список объектов и для каждого объекта – список допустимых операций, то такой способ хранения называется «мандаты» или «перечни возможностей» (capability list). Каждый пользователь обладает несколькими мандатами и может иметь право передавать их другим. Мандаты могут быть рассеяны по системе и вследствие этого представлять большую угрозу для безопасности, чем списки контроля доступа. Их хранение должно быть тщательно продумано.

Примерами систем, использующих перечни возможностей, являются Hydra, Cambridge CAP System [Denning, 1996].

Другие способы контроля доступа

Иногда применяется *комбинированный способ*. Например, в том же Unix на этапе открытия файла происходит анализ ACL (операция open). В случае благоприятного исхода файл заносится в список открытых процессом файлов, и при последующих операциях чтения и записи проверки прав доступа не происходит. Список открытых файлов можно рассматривать как перечень возможностей.

Существует также схема *lock-key*, которая является компромиссом между списками прав доступа и перечнями возможностей. В этой схеме каждый объект имеет список уникальных битовых шаблонов (patterns), называемых locks. Аналогично каждый домен имеет список уникальных битовых шаблонов, называемых ключами (keys). Процесс, выполняющийся в домене, может получить доступ к объекту, только если домен имеет ключ, который соответствует одному из шаблонов объекта.

Как и в случае мандатов, список ключей для домена должен управляться ОС. Пользователям не разрешается проверять или модифицировать списки ключей (или шаблонов) непосредственно. Более подробно данная схема изложена в [Silberschatz, 2002].

Смена домена

В большинстве ОС для определения домена применяются идентификаторы пользователей. Обычно переключение между доменами происходит, когда меняется пользователь. Но почти все системы нуждаются в дополнительных механизмах смены домена, которые используются, когда некая привилегированная возможность необходима большому количеству пользователей. Например, может понадобиться разрешить пользователям иметь доступ к сети, не заставляя их писать собственные сетевые программы. В таких случаях для процессов ОС Unix предусмотрена установка бита *set-uid*. В результате установки этого бита в сетевой программе она получает привилегии ее создателя (а не пользователя), заставляя домен меняться на время ее выполнения. Таким образом, рядовой пользователь может получить нужные привилегии для доступа к сети.

Недопустимость повторного использования объектов

Контроль повторного использования объекта предназначен для предотвращения попыток незаконного получения конфиденциальной информации, остатки которой могли сохраниться в некоторых объектах, ранее использовавшихся и освобожденных другим пользователем. Безопасность повторного применения должна гарантироваться для областей оперативной

памяти (в частности, для буферов с образами экрана, расшифрованными паролями и т. п.), для дисковых блоков и магнитных носителей в целом. Очистка должна производиться путем записи маскирующей информации в объект при его освобождении (перераспределении). Например, для дисков на практике применяется способ двойной перезаписи освободившихся после удаления файлов блоков случайной битовой последовательностью.

Выявление вторжений. Аудит системы защиты

Даже самая лучшая система защиты рано или поздно будет взломана. Обнаружение попыток вторжения является важнейшей задачей системы защиты, поскольку ее решение позволяет минимизировать ущерб от взлома и собирать информацию о методах вторжения. Как правило, поведение взломщика отличается от поведения легального пользователя. Иногда эти различия можно выразить количественно, например подсчитывая число некорректных вводов пароля во время регистрации.

Основным инструментом выявления вторжений является запись данных аудита. Отдельные действия пользователей протоколируются, а полученный протокол используется для выявления вторжений.

Аудит, таким образом, заключается в регистрации специальных данных о различных типах событий, происходящих в системе и так или иначе влияющих на состояние безопасности компьютерной системы. К числу таких событий обычно причисляют следующие:

- вход или выход из системы;
- операции с файлами (открыть, закрыть, переименовать, удалить);
- обращение к удаленной системе;
- смена привилегий или иных атрибутов безопасности (режима доступа, уровня благонадежности пользователя и т. п.).

Если фиксировать все события, объем регистрационной информации, скорее всего, будет расти слишком быстро, а ее эффективный анализ станет невозможным. Следует предусматривать наличие средств выборочного протоколирования как в отношении пользователей, когда слежение осуществляется только за подозрительными личностями, так и в отношении событий. Слежка важна в первую очередь как профилактическое средство. Можно надеяться, что многие воздержатся от нарушений безопасности, зная, что их действия фиксируются.

Помимо протоколирования, можно периодически *сканировать* систему на наличие слабых мест в системе безопасности. Такое сканирование может проверить разнообразные аспекты системы:

- короткие или легкие пароли;
- неавторизованные *set-uid* программы, если система поддерживает этот механизм;

- неавторизованные программы в системных директориях;
- долго выполняющиеся программы;
- нелогичная защита как пользовательских, так и системных директорий и файлов. Примером нелогичной защиты может быть файл, который запрещено читать его автору, но в который разрешено записывать информацию постороннему пользователю;
- потенциально опасные списки поиска файлов, которые могут привести к запуску «троянского коня»;
- изменения в системных программах, обнаруженные при помощи контрольных сумм.

Любая проблема, обнаруженная сканером безопасности, может быть как ликвидирована автоматически, так и передана для решения менеджеру системы.

Анализ некоторых популярных ОС с точки зрения их защищенности

Итак, ОС должна способствовать реализации мер безопасности или непосредственно поддерживать их. Примерами подобных решений в рамках аппаратуры и операционной системы могут быть:

- разделение команд по уровням привилегированности;
- сегментация адресного пространства процессов и организация защиты сегментов;
- защита различных процессов от взаимного влияния за счет выделения каждому своего виртуального пространства;
- особая защита ядра ОС;
- контроль повторного использования объекта;
- наличие средств управления доступом;
- структурированность системы, явное выделение надежной вычислительной базы (совокупности защищенных компонентов), обеспечение компактности этой базы;
- следование принципу минимизации привилегий – каждому компоненту дается ровно столько привилегий, сколько необходимо для выполнения им своих функций.

Большое значение имеет структура файловой системы. Например, в ОС с дискреционным контролем доступа каждый файл должен храниться вместе с дискреционным списком прав доступа к нему, а, например, при копировании файла все атрибуты, в том числе и ACL, должны быть автоматически скопированы вместе с телом файла.

В принципе, меры безопасности не обязательно должны быть заранее встроены в ОС – достаточно принципиальной возможности дополнительной установки защитных продуктов. Так, сугубо ненадежная система

MS-DOS может быть усовершенствована за счет средств проверки паролей доступа к компьютеру и/или жесткому диску, за счет борьбы с вирусами путем отслеживания попыток записи в загрузочный сектор CMOS-средствами и т. п. Тем не менее по-настоящему надежная система должна изначально проектироваться с акцентом на механизмы безопасности.

MS-DOS

ОС MS-DOS функционирует в реальном режиме (real-mode) процессора i80×86. В ней невозможно выполнение требования, касающегося изоляции программных модулей (отсутствует аппаратная защита памяти). Уязвимым местом для защиты является также файловая система FAT, не предполагающая у файлов наличия атрибутов, связанных с разграничением доступа к ним. Таким образом, MS-DOS находится на самом нижнем уровне в иерархии защищенных ОС.

NetWare, IntranetWare

Замечание об отсутствии изоляции модулей друг от друга справедливо и в отношении рабочей станции NetWare. Однако NetWare – *сетевая* ОС, поэтому к ней возможно применение и иных критериев. Это на данный момент единственная сетевая ОС, сертифицированная по классу С2 (следующей, по-видимому, будет Windows 2000). При этом важно изолировать наиболее уязвимый участок системы безопасности NetWare – консоль сервера, и тогда следование определенной практике поможет увеличить степень защищенности данной сетевой операционной системы. Возможность создания безопасных систем обусловлена тем, что число работающих приложений *фиксировано* и пользователь не имеет возможности запуска своих приложений.

OS/2

OS/2 работает в защищенном режиме (protected-mode) процессора i80×86. Изоляция программных модулей реализуется при помощи встроенных в этот процессор механизмов защиты памяти. Поэтому она свободна от указанного выше коренного недостатка систем типа MS-DOS. Но OS/2 была спроектирована и разработана без учета требований по защите от несанкционированного доступа. Это сказывается прежде всего на файловой системе. В файловых системах OS/2 HPFS (High Performance File System) и FAT нет места ACL. Кроме того, пользовательские программы имеют возможность запрета прерываний. Следовательно, сертификация OS/2 на соответствие какому-то классу защиты не представляется возможной.

Считается, что такие операционные системы, как MS-DOS, Mac OS, Windows, OS/2, имеют уровень защищенности D (по оранжевой книге). Но, если быть точным, нельзя считать эти ОС даже системами уровня безопасности D, ведь они никогда не представлялись на тестирование.

Unix

Рост популярности Unix и все большая осведомленность о проблемах безопасности привели к осознанию необходимости достижения приемлемого уровня безопасности ОС, сохранив при этом мобильность, гибкость и открытость программных продуктов. В Unix есть несколько уязвимых с точки зрения безопасности мест, хорошо известных опытным пользователям, вытекающих из самой природы Unix (см., например, раздел «Типичные объекты атаки хакеров» в книге [Дунаев, 1996]). Однако хорошее системное администрирование может ограничить эту уязвимость.

Относительно защищенности Unix сведения противоречивы. В Unix изначально были заложены идентификация пользователей и разграничение доступа. Как оказалось, средства защиты данных в Unix могут быть доработаны, и сегодня можно утверждать, что многие клоны Unix по всем параметрам соответствуют классу безопасности C2.

Обычно, говоря о защищенности Unix, рассматривают защищенность автоматизированных систем, одним из компонентов которых является Unix-сервер. Безопасность такой системы увязывается с защитой глобальных и локальных сетей, безопасностью удаленных сервисов типа telnet и rlogin/rsh и аутентификацией в сетевой конфигурации, безопасностью X Window-приложений. На системном уровне важно наличие средств идентификации и аудита.

В Unix существует список именованных пользователей, в соответствии с которым может быть построена система разграничения доступа.

В ОС Unix считается, что информация, нуждающаяся в защите, находится главным образом в файлах.

По отношению к конкретному файлу все пользователи делятся на три категории:

- владелец файла;
- члены группы владельца;
- прочие пользователи.

Для каждой из этих категорий режим доступа определяет права на операции с файлом, а именно:

- право на чтение;
- право на запись;
- право на выполнение (для каталогов – право на поиск).

В итоге девяти (3×3) битов защиты оказывается достаточно, чтобы специфицировать ACL каждого файла.

Аналогичным образом защищены и другие объекты ОС Unix, например семафоры, сегменты разделяемой памяти и т. п.

Указанных видов прав достаточно, чтобы определить допустимость любой операции с файлами. Например, для удаления файла необходимо иметь право на запись в соответствующий каталог. Как уже говорилось, права доступа к файлу проверяются только на этапе открытия. При последующих операциях чтения и записи проверка не выполняется. В результате, если режим доступа к файлу меняется после того, как файл был открыт, это не сказывается на процессах, уже открывших этот файл. Данное обстоятельство является уязвимым с точки зрения безопасности местом.

Наличие всего трех видов субъектов доступа — владелец, группа, все остальные – затрудняет задание прав «с точностью до пользователя», особенно в случае больших конфигураций. В популярной разновидности Unix – Solaris имеется возможность использования списков управления доступом (ACL), позволяющих индивидуально устанавливать права доступа отдельных пользователей или групп.

Среди всех пользователей особое положение занимает пользователь root, обладающий максимальными привилегиями. Обычные правила разграничения доступа к нему не применяются – ему доступна вся информация на компьютере.

В Unix имеются инструменты системного аудита – хронологическая запись событий, имеющих отношение к безопасности. К таким событиям обычно относят: обращения программ к отдельным серверам; события, связанные с входом/выходом в систему и другие. Обычно регистрационные действия выполняются специализированным syslog-демоном, который проводит запись событий в регистрационный журнал в соответствии с текущей конфигурацией. Syslog-демон стартует в процессе загрузки системы.

Таким образом, безопасность ОС Unix может быть доведена до соответствия классу C2. Однако разработка на ее основе автоматизированных систем более высокого класса защищенности может быть сопряжена с большими трудозатратами.

Windows NT/2000/XP

С момента выхода версии 3.1 осенью 1993 года в Windows NT гарантировалось соответствие уровню безопасности C2. В настоящее время (точнее, в 1999 году) сертифицирована версия NT 4 с Service Pack 6a с использованием файловой системы NTFS в автономной и сетевой конфигурациях. Следует помнить, что этот уровень безопасности не подразумевает

защиту информации, передаваемой по сети, и не гарантирует защищенности от физического доступа.

Компоненты защиты NT частично встроены в ядро, а частично реализуются подсистемой защиты. Подсистема защиты контролирует доступ и учетную информацию. Кроме того, Windows NT имеет встроенные средства, такие как поддержка резервных копий данных и управление источниками бесперебойного питания, которые не требуются оранжевой книгой, но в целом повышают общий уровень безопасности.

ОС Windows 2000 сертифицирована по стандарту Common Criteria. В дальнейшем линейку продуктов Windows NT/2000/XP, изготовленных по технологии NT, будем называть просто Windows NT.

Ключевая цель системы защиты Windows NT – следить за тем, кто и к каким объектам осуществляет доступ. Система защиты хранит информацию, относящуюся к безопасности для каждого пользователя, группы пользователей и объекта. Единообразие контроля доступа к различным объектам (процессам, файлам, семафорам и др.) обеспечивается тем, что с каждым процессом связан маркер доступа, а с каждым объектом – дескриптор защиты. Маркер доступа в качестве параметра имеет идентификатор пользователя, а дескриптор защиты – списки прав доступа. ОС может контролировать попытки доступа, которые производятся процессами прямо или косвенно инициированными пользователем.

Windows NT отслеживает и контролирует доступ как к объектам, которые пользователь может видеть посредством интерфейса (такие, как файлы и принтеры), так и к объектам, которые пользователь не может видеть (например, процессы и именованные каналы). Любопытно, что, помимо разрешающих записей, списки прав доступа содержат и запрещающие записи, чтобы пользователь, которому доступ к какому-либо объекту запрещен, не смог получить его как член какой-либо группы, которой этот доступ предоставлен.

Система защиты ОС Windows NT состоит из следующих компонентов:

- Процедуры регистрации (Logon Processes), которые обрабатывают запросы пользователей на вход в систему. Они включают в себя начальную интерактивную процедуру, отображающую начальный диалог с пользователем на экране и удаленные процедуры входа, которые позволяют удаленным пользователям получить доступ с рабочей станции сети к серверным процессам Windows NT.
- Подсистемы локальной авторизации (Local Security Authority, LSA), которая гарантирует, что пользователь имеет разрешение на доступ в систему. Этот компонент – центральный для системы защиты Windows NT. Он порождает маркеры доступа, управляет локальной политикой безопасности и предоставляет интерактивным пользователям аутентификационные услуги. LSA также контролирует политику

аудита и ведет журнал, в котором сохраняются сообщения, порождаемые диспетчером доступа.

- Менеджера учета (Security Account Manager, SAM), который управляет базой данных учета пользователей. Эта база данных содержит информацию обо всех пользователях и группах пользователей. SAM предоставляет услуги по легализации пользователей, применяющиеся в LSA.
- Диспетчера доступа (Security Reference Monitor, SRM), который проверяет, имеет ли пользователь право на доступ к объекту и на выполнение тех действий, которые он пытается совершить. Этот компонент обеспечивает легализацию доступа и политику аудита, определяемые LSA. Он предоставляет услуги для программ супервизорного и пользовательского режимов, для того чтобы гарантировать, что пользователи и процессы, осуществляющие попытки доступа к объекту, имеют необходимые права. Данный компонент также порождает сообщения службы аудита, когда это необходимо.

Microsoft Windows NT – относительно новая ОС, которая была спроектирована для поддержки разнообразных защитных механизмов, от минимальных до C2, и безопасность которой наиболее продумана. Дефолтный уровень называется минимальным, но он легко может быть доведен системным администратором до желаемого уровня.

Заключение

Решение вопросов безопасности операционных систем обусловлено их архитектурными особенностями и связано с правильной организацией идентификации и аутентификации, авторизации и аудита.

Наиболее простой подход к аутентификации – применение пользовательского пароля. Пароли уязвимы, значительная часть попыток несанкционированного доступа в систему связана с компрометацией паролей.

Авторизация связана со специфицированием совокупности аппаратных и программных объектов, нуждающихся в защите. Для защиты объекта устанавливаются права доступа к нему. Набор прав доступа определяет домен безопасности. Формальное описание модели защиты осуществляется с помощью матрицы доступа, которая может храниться в виде списков прав доступа или перечней возможностей.

Аудит системы заключается в регистрации специальных данных о различных событиях, происходящих в системе и так или иначе влияющих на состояние безопасности компьютерной системы.

Среди современных ОС вопросы безопасности лучше всего продуманы в ОС Windows NT.

Семинары

Семинары 1–2. Введение в курс практических занятий. Знакомство с операционной системой UNIX

Введение в курс практических занятий. Краткая история операционной системы UNIX, ее структура. Системные вызовы и библиотека libc. Понятия login и password. Упрощенное понятие об устройстве файловой системы в UNIX. Полные имена файлов. Понятие о текущей директории. Команда pwd. Относительные имена файлов. Домашняя директория пользователя и ее определение. Команда man – универсальный справочник. Команды cd – смены текущей директории и ls – просмотра состава директории. Команда cat и создание файла. Перенаправление ввода и вывода. Простейшие команды для работы с файлами – cp, rm, mkdir, mv. История редактирования файлов – ed, vi. Система Midnight Commander – mc. Встроенный mc редактор и редактор joe. Пользователь и группа. Команды chown и chgrp. Права доступа к файлу. Команда ls с опциями -al. Использование команд chmod и umask. Системные вызовы getuid и getgid. Компиляция программ на языке С в UNIX и запуск их на счет.

Ключевые слова: системный вызов, переменная errno, функция perror, login, password, полное имя файла, относительное имя файла, домашняя директория, текущая директория, команды man, cd, ls, cp, mv, rm, mkdir, cat, chown, chgrp, chmod, umask, маска создания файлов текущего процесса, идентификатор пользователя, UID, идентификатор группы пользователей, GID, системные вызовы getuid, getgid.

Введение в курс практических занятий

Настоящий курс практических занятий является одной из первых известных авторам попыток систематически проиллюстрировать лекционный курс «Основы операционных систем» на примере конкретной операционной системы, а именно – операционной системы UNIX.

Необходимость связывания систематического изложения материала семинарских и практических занятий с материалом лекций возникла в процессе становления базового четырехсеместрового набора курсов по информатике в МФТИ. Новизна излагаемого материала для многих преподавателей заставила лекторов для соблюдения некоторого стандарта обучения готовить по каждой теме методические указания для

участников семинаров, и эти указания, как показал опыт, могут с успехом использоваться и студентами, в том числе для самостоятельного обучения. Обкатанная версия этих методических указаний — расширенная, дополненная и модифицированная по результатам многочисленных обсуждений — предлагается сейчас вашему вниманию.

Семестровый курс «Основы операционных систем» является третьим по счету курсом цикла «Информатика», которому предшествуют курсы «Алгоритмы и алгоритмические языки» и «Архитектура ЭВМ и язык Ассемблера». Предполагается, что к началу практических занятий студенты умеют программировать на языке С (с использованием функций стандартной библиотеки для работы с файлами и строками) и имеют представление о внутреннем устройстве ЭВМ.

Переход от обучения студентов информатике с использованием мэйнфреймов к обучению с использованием сетевых классов персональных компьютеров, свершившийся за последние двадцать лет, неизбежно наложил свой отпечаток на форму проведения практических занятий. Вместо раздельного проведения семинаров и практикума (лабораторных работ) появилось нечто смешанное – семинарский практикум или практический семинар, когда изложение нового материала в течение одного занятия чередуется с короткими практическими программными работами. Именно в виде таких семинаров-практикумов и построен наш курс. Ввиду достаточно высокой сложности используемых программных конструкций мы решили приводить готовые примеры программ для иллюстрации рассматриваемых понятий с последующей их модификацией студентами. Это позволило увеличить насыщенность занятий и за семестровый курс охватить большее количество материала.

Для иллюстрации лекций была выбрана операционная система UNIX, как наиболее открытая, изящная и простая для понимания, хотя создание подобного практического курса возможно и для других операционных систем, например для Windows NT.

В целом практический курс включает в себя 16 занятий, одно из которых в середине семестра – между семинарами 9 и 10-11 – посвящено проведению контрольной работы по материалам лекций. Некоторым темам выделено по два занятия, и соответствующие семинары имеют сдвоенные номера. Естественно, разбиение тем на занятия является достаточно условным; желательно лишь, чтобы они непосредственно следовали за лекциями, на которых основываются.

Далее мы переходим к изложению материала семинарско-практического курса.

По своему содержанию материал текущих семинаров 1–2 является наиболее критичным по отношению к используемому виду операционной системы и политике администрирования. Поэтому многие вопросы

будут содержать ссылку **«узнайте у своего системного администратора»**. Прежде чем приступать к занятиям, необходимо обеспечить наличие пользовательских аккаунтов для обучающихся. **«Узнайте у своего системного администратора»**, как это сделать.

В тексте семинаров программные конструкции, включая имена системных вызовов, стандартных функций и команды оболочки операционной системы, `выделены другим шрифтом`. В UNIX системные вызовы и команды оболочки инициируют сложные последовательности действий, затрагивая различные аспекты функционирования операционной системы. Как правило, в рамках одного семинара полное объяснение всех нюансов их поведения является невозможным. Поэтому подробные описания большинства используемых системных вызовов, системных функций и некоторых команд оболочки операционной системы при первой встрече с ними вынесены из основного текста на серый фон и обведены рамочкой, а в основном тексте рассматриваются только те детали их описания, для понимания которых хватает накопленных знаний.

Если какой-либо параметр у команды оболочки является необязательным, он будет указываться в квадратных скобках, например, `[who]`. В случае, когда возможен выбор только одного из нескольких возможных вариантов параметров, варианты будут перечисляться в фигурных скобках и разделяться вертикальной чертой, например, `{+ | - | =}`.

Краткая история операционной системы UNIX, ее структура

На первой лекции мы разобрали содержание понятия «операционная система», обсудили функции операционных систем и способы их построения. Все материалы первой и последующих лекций мы будем иллюстрировать практическими примерами, связанными с использованием одной из разновидностей операционной системы UNIX – операционной системы Linux, хотя постараемся не связывать свой рассказ именно с ее особенностями.

Ядро операционной системы Linux представляет собой монолитную систему. При компиляции ядра Linux можно разрешить динамическую загрузку и выгрузку очень многих компонентов ядра – так называемых модулей. В момент загрузки модуля его код загружается для исполнения в привилегированном режиме и связывается с остальной частью ядра. Внутри модуля могут использоваться любые экспортируемые ядром функции.

Свой нынешний вид эта операционная система обрела в результате длительной эволюции UNIX-образных операционных систем. История развития UNIX подробно освещена практически во всей литературе, посвященной вычислительной технике. Как правило, это во многом один и

тот же текст, с небольшими изменениями кочующий из одного издания в другое, и нам не хотелось бы повторяться. Мы просто сошлемся на достаточно подробное изложение в книге [Олифер, 2001] или на оригинальную работу одного из родоначальников UNIX [Ritchie, 1984]. Для нас наиболее важным во всей этой истории является существование двух стержневых линий эволюции – линии System V и линии BSD, поскольку в процессе обучения мы будем сталкиваться с различиями в их реализации.

Системные вызовы и библиотека libc

Основной постоянно функционирующей частью операционной системы UNIX является ее ядро. Другие программы (системные или пользовательские) могут общаться с ядром посредством системных вызовов, которые по сути дела являются прямыми точками входа программ в ядро. При исполнении системного вызова программа пользователя временно переходит в привилегированный режим, получая доступ к данным или устройствам, которые недоступны при работе в режиме пользователя.

Реальные машинные команды, необходмые для активизации системных вызовов, естественно, отличаются от машины к машине, наряду со способом передачи параметров и результатов между вызывающей программой и ядром. Однако с точки зрения программиста на языке С использование системных вызовов ничем внешне не отличается от использования других функций стандартной ANSI библиотеки языка С, таких как функции работы со строками `strlen()`, `strcpy()` и т. д. Стандартная библиотека UNIX – `libc` – обеспечивает С-интерфейс к каждому системному вызову. Это приводит к тому, что системный вызов выглядит как функция на языке С для программиста. Более того, многие из уже известных вам стандартных функций, например функции для работы с файлами: `fopen()`, `fread()`, `fwrite()` при реализации в операционной системе UNIX будут применять различные системные вызовы. По ходу курса нам придется познакомиться с большим количеством разнообразных системных вызовов и их С-интерфейсами.

Большинство системных вызовов, возвращающих целое значение, использует значение `-1` для оповещения о возникновении ошибки и значение большее или равное `0` – при нормальном завершении. Системные вызовы, возвращающие указатели, обычно для идентификации ошибочной ситуации пользуются значением `NULL`. Для точного определения причины ошибки С-интерфейс предоставляет глобальную переменную `errno`, описанную в файле `<errno.h>` вместе с ее возможными значениями и их краткими определениями. Заметим, что анализировать значение переменной `errno` необходимо сразу после возникновения ошибочной ситуации, так как успешно завершившиеся системные вызовы не изменяют ее значения. Для получения символьной информации об ошибке на

стандартном выводе программы для ошибок (по умолчанию экран терминала) может применяться стандартная UNIX-функция `perror()`.

Функция perror()

Прототип функции

```
#include <stdio.h>
void perror(char *str);
```

Описание функции

Функция `perror` предназначена для вывода сообщения об ошибке, соответствующего значению системной переменной `errno` на стандартный поток вывода ошибок. Функция печатает содержимое строки `str` (если параметр `str` не равен `NULL`), двоеточие, пробел и текст сообщения, соответствующий возникшей ошибке, с последующим символом перевода строки (`'\n'`).

Понятия login и password

Операционная система UNIX является многопользовательской операционной системой. Для обеспечения безопасной работы пользователей и целостности системы доступ к ней должен быть санкционирован. Для каждого пользователя, которому разрешен вход в систему, заводится специальное регистрационное имя – username или login и сохраняется специальный пароль – password, соответствующий этому имени. Как правило, при заведении нового пользователя начальное значение пароля для него задает системный администратор. После первого входа в систему пользователь должен изменить начальное значение пароля с помощью специальной команды. В дальнейшем он может в любой момент изменить пароль по своему желанию.

«Узнайте у своего системного администратора» регистрационные имена и пароли, установленные для обучающихся.

Вход в систему и смена пароля

Настало время первый раз войти в систему. Если в системе установлена графическая оболочка наряду с обычными алфавитно-цифровыми терминалами, лучше всего это сделать с алфавитно-цифрового терминала или его эмулятора. На экране появляется надпись, предлагающая ввести регистрационное имя, как правило, это «login:». Набрав свое регистрационное имя, нажмите клавишу <Enter>. Система запросит у вас пароль, соответствующий введенному имени, выдав специальное приглашение – обычно «Password:». Внимательно наберите пароль, установленный для

вас системным администратором, и нажмите клавишу <Enter>. **Вводимый пароль на экране не отображается, поэтому набирайте его аккуратно!** Если все было сделано правильно, у вас на экране появится приглашение к вводу команд операционной системы.

Пароль, установленный системным администратором, необходимо сменить. **«Узнайте у своего системного администратора»**, какая команда для этого используется на вашей вычислительной системе (чаще всего это команда passwd или yppasswd). В большинстве UNIX-образных систем требуется, чтобы новый пароль имел не менее шести символов и содержал, по крайней мере, две не буквы и две не цифры. **«Узнайте у своего системного администратора»**, какие ограничения на новый пароль существуют в вашей операционной системе.

Придумайте новый пароль и хорошенько его запомните, а лучше запишите. Пароли в операционной системе хранятся в закодированном виде, и если вы его забыли, никто не сможет помочь вам его вспомнить. Единственное, что может сделать системный администратор, так это установить вам новый пароль. **«Узнайте у своего системного администратора»**, что нужно предпринять, если вы забыли пароль.

Введите команду для смены пароля. Обычно система просит сначала набрать старый пароль, затем ввести новый и подтвердить правильность его набора повторным введением. После смены пароля уже никто посторонний не сможет войти в систему под вашим регистрационным именем.

Congratulations!!! Теперь вы полноценный пользователь операционной системы UNIX.

Упрощенное понятие об устройстве файловой системы в UNIX. Полные и относительные имена файлов

В операционной системе UNIX существует три базовых понятия: *«процесс»*, *«файл»* и *«пользователь»*. С понятием «пользователь» мы только что уже столкнулись и будем сталкиваться в дальнейшем при изучении работы операционной системы UNIX. Понятие «процесс» характеризует динамическую сторону происходящего в вычислительной системе, оно будет подробно обсуждаться в лекции 2 и в описании последующих семинаров. Понятие «файл» характеризует статическую сторону вычислительной системы.

Из предыдущего опыта работы с вычислительной техникой вы уже имеете некоторое представление о файле как об именованном наборе данных, хранящемся где-нибудь на магнитных дисках или лентах. Для нашего сегодняшнего обсуждения нам достаточно такого понимания, чтобы

разобраться в том, как организована работа с файлами в операционной системе UNIX. Более подробное рассмотрение понятия «файл» и организации файловых систем для операционных систем в целом будет приведено в лекциях 11–12, а также на семинарах 11–12, посвященных организации файловых систем в UNIX.

Все файлы, доступные в операционной системе UNIX, как и в уже известных вам операционных системах, объединяются в древовидную логическую структуру. Файлы могут объединяться в *каталоги* или *директории*. Не существует файлов, которые не входили бы в состав какой-либо директории. Директории в свою очередь могут входить в состав других директорий. Допускается существование пустых директорий, в которые не входит ни один файл и ни одна другая директория (см. рис. 1–2.1). Среди всех директорий существует только одна директория, которая не входит в состав других директорий – ее принято называть *корневой*. На настоящем уровне нашего незнания UNIX мы можем заключить, что в файловой системе UNIX присутствует, по крайней мере, два типа файлов: обычные файлы, которые могут содержать тексты программ, исполняемый код, данные и т. д. – их принято называть *регулярными файлами*, и директории.

Каждому файлу (регулярному или директории) должно быть присвоено имя. В различных версиях операционной системы UNIX существуют те или иные ограничения на построение имени файла. В стандарте POSIX на интерфейс системных вызовов для операционной системы UNIX содержится лишь три явных ограничения:

- Нельзя создавать имена большей длины, чем это предусмотрено операционной системой (для Linux – 255 символов).

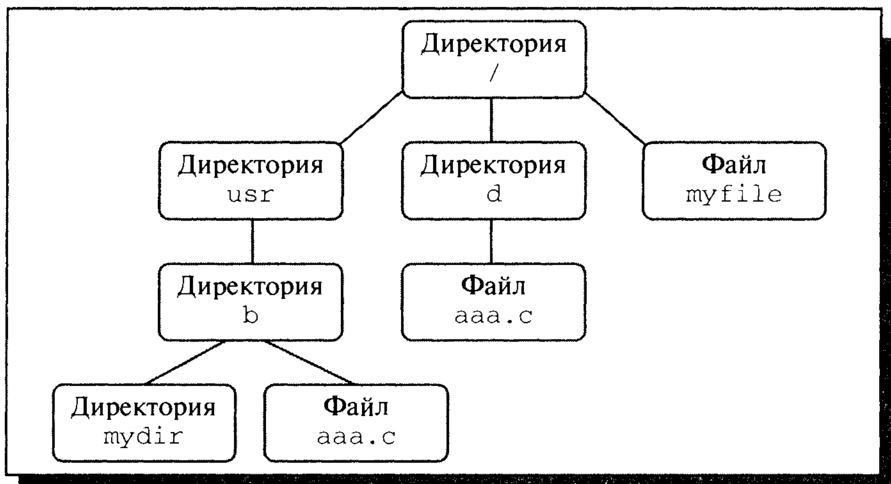


Рис. 1–2.1. Пример структуры файловой системы

- Нельзя использовать символ NUL (не путать с указателем NULL!) – он же символ с нулевым кодом, он же признак конца строки в языке С.
- Нельзя использовать символ '/'.

От себя добавим, что также нежелательно применять символы «звездочка» – «*», «знак вопроса» – «?», «кавычка» – «\"», «апостроф» – «\'», «пробел» – « » и «обратный слэш» – «\\» (символы записаны в нотации символьных констант языка С).

Единственным исключением является корневая директория, которая **всегда** имеет имя «/». Эта же директория по вполне понятным причинам представляет собой единственный файл, который должен иметь уникальное имя во всей файловой системе. Для всех остальных файлов имена должны быть уникальными только в рамках той директории, в которую они непосредственно входят. Каким же образом отличить два файла с именами "aaa.c", входящими в директории "b" и "c" на рисунке 1–2.1, чтобы было понятно, о каком из них идет речь? Здесь на помощь приходит *понятие полного имени файла*.

Давайте мысленно построим путь от корневой вершины дерева файлов к интересующему нас файлу и выпишем все имена файлов (т. е. узлов дерева), встречающиеся на нашем пути, например, "/ usr b aaa.c"». В этой последовательности первым будет всегда стоять имя корневой директории, а последним – имя интересующего нас файла. Отделим имена узлов друг от друга в этой записи не пробелами, а символами «/», за исключением имени корневой директории и следующего за ним имени ("/usr/b/aaa.c"). Полученная запись однозначно идентифицирует файл во всей логической конструкции файловой системы. Такая запись и получила название полного имени файла.

Понятие о текущей директории. Команда pwd. Относительные имена файлов

Полные имена файлов могут включать в себя достаточно много имен директорий и быть очень длинными, с ними не всегда удобно работать. В то же время, существуют такие понятия как *текущая* или *рабочая директория* и *относительное имя файла*.

Для каждой работающей программы в операционной системе, включая командный интерпретатор (shell), который обрабатывает вводимые команды и высвечивает приглашение к их вводу, одна из директорий в логической структуре файловой системы назначается текущей или рабочей для данной программы. Узнать, какая директория является текущей для вашего командного интерпретатора, можно с помощью команды операционной системы pwd.

Команда pwd

Синтаксис команды

```
pwd
```

Описание команды

Команда pwd **выводит полное имя текущей директории для работающего командного интерпретатора.**

Зная текущую директорию, мы можем проложить путь по графу файлов от текущей директории к интересующему нас файлу. Запишем последовательность узлов, которые встретятся на этом пути, следующим образом. Узел, соответствующий текущей директории, в запись не включаем. При движении по направлению к корневому каталогу каждый узел будем обозначать двумя символами «точка» – «..», а при движении по направлению от корневого каталога будем записывать имя встретившегося узла. Разделим обозначения, относящиеся к разным узлам в этой записи, символами «/». Полученную строку принято называть относительным именем файла. Относительные имена файлов меняются при смене рабочего каталога. Так, в нашем примере, если рабочий каталог – это директория "/d", то для файла "/usr/b/aaa.c" относительным именем будет "../usr/b/aaa.c", а если рабочий каталог – это директория "/usr/b", то его относительное имя – "aaa.c".

Для полноты картины имя текущего каталога можно вставлять в относительное имя файла, обозначая текущий каталог одиночным символом «точка» – «.». Тогда наши относительные имена будут выглядеть как "./../usr/b/aaa.c" и "./aaa.c" соответственно.

Программы, запущенные с помощью командного интерпретатора, будут иметь в качестве рабочей директории его рабочую директорию, если внутри этих программ не изменить ее расположение с помощью специального системного вызова.

Домашняя директория пользователя и ее определение

Для каждого нового пользователя в системе заводится специальная директория, которая становится текущей сразу после его входа в систему. Эта директория получила название *домашней директории* пользователя. Воспользуйтесь командой pwd для определения своей домашней директории.

Команда man – универсальный справочник

По ходу изучения операционной системы UNIX вам часто будет требоваться информация о том, что делает та или иная команда или системный вызов, какие у них параметры и опции, для чего предназначены некоторые системные файлы, каков их формат и т. д. Мы постарались, по мере возможности, включить описания большинства используемых в курсе команд и системных вызовов в наш текст. Однако иногда для получения более полной информации мы отсылаем читателей к UNIX Manual – руководству по операционной системе UNIX. К счастью, большая часть информации в UNIX Manual доступна в интерактивном режиме с помощью утилиты man.

Пользоваться утилитой man достаточно просто – наберите команду

```
man имя
```

где имя – это имя интересующей вас команды, утилиты, системного вызова, библиотечной функции или файла. Попробуйте с ее помощью посмотреть информацию о команде pwd.

Чтобы пролистать страницу полученного описания, если оно не поместилось на экране полностью, следует нажать клавишу <пробел>. Для прокрутки одной строки воспользуйтесь клавишей <Enter>. Вернуться на страницу назад позволит одновременное нажатие клавиш <Ctrl> и <b>. Выйти из режима просмотра информации можно с помощью клавиши <q>.

Иногда имена команд интерпретатора и системных вызовов или какие-либо еще имена совпадают. Тогда чтобы найти интересующую вас информацию, необходимо задать утилите man категорию, к которой относится эта информация (номер раздела). Деление информации по категориям может слегка отличаться от одной версии UNIX к другой. В Linux, например, принято следующее разделение:

1. Исполняемые файлы или команды интерпретатора.
2. Системные вызовы.
3. Библиотечные функции.
4. Специальные файлы (обычно файлы устройств) – что это такое, вы узнаете на семинарах 13–14.
5. Формат системных файлов и принятые соглашения.
6. Игры (обычно отсутствуют).
7. Макропакеты и утилиты – такие как сам man.
8. Команды системного администратора.
9. Подпрограммы ядра (нестандартный раздел).

Если вы знаете раздел, к которому относится информация, то утилиту man можно вызвать в Linux с дополнительным параметром

```
man номер_раздела имя
```

В других операционных системах этот вызов может выглядеть иначе. Для получения точной информации о разбиении на разделы, форме указания номера раздела и дополнительных возможностях утилиты man наберите команду

```
man man
```

Команды cd – для смены текущей директории и ls – для просмотра состава директории

Для смены текущей директории командного интерпретатора можно воспользоваться командой cd (change directory). Для этого необходимо набрать команду в виде

```
cd имя_директории
```

где имя_директории – полное или относительное имя директории, которую вы хотите сделать текущей. Команда cd без параметров сделает текущей директорией вашу домашнюю директорию.

Просмотреть содержимое текущей или любой другой директории можно, воспользовавшись командой ls (от list). Если ввести ее без параметров, эта команда распечатает вам список файлов, находящихся в текущей директории. Если же в качестве параметра задать полное или относительное имя директории

```
ls имя_директории
```

то она распечатает список файлов в указанной директории. Надо отметить, что в полученный список не войдут файлы, имена которых начинаются с символа «точка» – «.». Такие файлы обычно создаются различными системными программами для своих целей (например, для настройки). Посмотреть полный список файлов можно, дополнительно указав команде ls опцию -a, т.е. набрав ее в виде

```
ls -a
```

или

```
ls -a имя_директории
```

У команды ls существует и много других опций, часть из которых мы еще рассмотрим на семинарах. Для получения полной информации о команде ls воспользуйтесь утилитой man.

Путешествие по структуре файловой системы

Пользуясь командами cd, ls и pwd, попутешествуйте по структуре файловой системы и просмотрите ее содержимое. Возможно, зайти в некоторые директории или посмотреть их содержимое вам не удастся. Это связано с работой механизма защиты файлов и директорий, о котором мы поговорим позже. Не забудьте в конце путешествия вернуться в свою домашнюю директорию.

Команда cat и создание файла. Перенаправление ввода и вывода

Мы умеем перемещаться по логической структуре файловой системы и рассматривать ее содержимое. Хотелось бы уметь еще и просматривать содержимое файлов, и создавать их. Для просмотра содержимого небольшого текстового файла на экране можно воспользоваться командой cat. Если набрать ее в виде

```
cat имя_файла
```

то на экран выплеснется все его содержимое.

Внимание! Не пытайтесь рассматривать на экране содержимое директорий – все равно не получится! Не пытайтесь просматривать содержимое неизвестных файлов, особенно если вы не знаете, текстовый он или бинарный. Вывод на экран бинарного файла может привести к непредсказуемому поведению вашего терминала.

Если даже ваш файл и текстовый, но большой, то все равно вы увидите только его последнюю страницу. Большой текстовый файл удобнее рассматривать с помощью утилиты more (описание ее использования вы найдете в UNIX Manual). Команда cat будет нам интересна с другой точки зрения.

Если мы в качестве параметров для команды cat зададим не одно имя, а имена нескольких файлов

```
cat файл1 файл2 ... файлN
```

то система выдаст на экран их содержимое в указанном порядке. Вывод команды cat можно перенаправить с экрана терминала в какой-нибудь

файл, воспользовавшись символом перенаправления выходного потока данных – знаком «больше» – «>». Команда

```
cat файл1 файл2 ... файлN > файл_результата
```

сольет содержимое всех файлов, чьи имена стоят перед знаком «>», воедино в `файл_результата` – конкатенирует их (от слова concatenate и произошло ее название). Прием перенаправления выходных данных со стандартного потока вывода (экрана) в файл является стандартным для всех команд, выполняемых командным интерпретатором. Вы можете получить файл, содержащий список всех файлов текущей директории, если выполните команду `ls -a` с перенаправлением выходных данных:

```
ls -a > новый_файл
```

Если имена входных файлов для команды `cat` не заданы, то она будет использовать в качестве входных данных информацию, которая вводится с клавиатуры, до тех пор, пока вы не наберете признак окончания ввода – комбинацию клавиш <CTRL> и <d>.

Таким образом, команда

```
cat > новый_файл
```

позволяет создать новый текстовый файл с именем `новый_файл` и содержимым, которое пользователь введет с клавиатуры. У команды `cat` существует множество различных опций. Посмотреть ее полное описание можно в UNIX Manual.

Заметим, что наряду с перенаправлением выходных данных существует способ перенаправить входные данные. Если во время выполнения некоторой команды требуется ввести данные с клавиатуры, можно положить их заранее в файл, а затем перенаправить стандартный ввод этой команды с помощью знака «меньше» – «<» и следующего за ним имени файла с входными данными. Другие варианты перенаправления потоков данных можно посмотреть в UNIX Manual для командного интерпретатора.

Создание файла с помощью команды cat

Убедитесь, что вы находитесь в своей домашней директории, и создайте с помощью команды `cat` новый текстовый файл. Просмотрите его содержимое.

Простейшие команды работы с файлами – cp, rm, mkdir, mv

Для нормальной работы с файлами необходимо не только уметь создавать файлы, просматривать их содержимое и перемещаться по логическому дереву файловой системы. Нужно уметь создавать собственные поддиректории, копировать и удалять файлы, переименовывать их. Это минимальный набор операций, не владея которым нельзя чувствовать себя уверенно при работе с компьютером.

Для создания новой поддиректории используется команда mkdir (сокращение от make directory). В простейшем виде команда выглядит следующим образом:

```
mkdir имя_директории
```

где имя_директории – полное или относительное имя создаваемой директории. У команды mkdir имеется набор опций, описание которых можно просмотреть с помощью утилиты man.

Команда cp

Синтаксис команды

```
cp файл_источник файл_назначения
cp файл1 файл2 ... файлN дир_назначения
cp -r дир_источник дир_назначения
cp -r дир1 дир2 ... дирN дир_назначения
```

Описание команды

Настоящее описание является не полным описанием команды cp, а кратким введением в ее использование. Для получения полного описания команды обратитесь к UNIX Manual.

Команда cp в форме

```
cp файл_источник файл_назначения
```

служит для копирования одного файла с именем файл_источник в файл с именем файл_назначения.

Команда cp в форме

```
cp файл1 файл2 ... файлN дир_назначения
```

служит для копирования файла или файлов с именами файл1, файл2, ... файлN в уже существующую директорию с именем дир_назначения под своими именами. Вместо имен копируемых файлов могут использоваться их шаблоны.

Команда cp **в форме**

```
cp -r дир_источник дир_назначения
```

служит для рекурсивного копирования одной директории с именем дир_источник **в новую директорию с именем** дир_назначения. **Если директория** дир_назначения **уже существует, то мы получаем команду** cp **в следующей форме:**

```
cp -r дир1 дир2 ... дирN дир_назначения
```

Такая команда служит для рекурсивного копирования директории или директорий с именами дир1, дир2, ... дирN **в уже существующую директорию с именем** дир_назначения **под своими собственными именами. Вместо имен копируемых директорий могут использоваться их шаблоны.**

Для копирования файлов может использоваться команда cp (сокращение от copy). Команда cp умеет копировать не только отдельный файл, но и набор файлов, и даже директорию целиком вместе со всеми входящими в нее поддиректориями (рекурсивное копирование). Для задания набора файлов могут использоваться шаблоны имен файлов. Точно так же шаблон имени может быть использован и в командах переименования файлов и их удаления, которые мы рассмотрим ниже.

Шаблоны имен файлов

Шаблоны имен файлов могут применяться в качестве параметра для задания набора имен файлов во многих командах операционной системы. При использовании шаблона просматривается вся совокупность имен файлов, находящихся в файловой системе, и те имена, которые удовлетворяют шаблону, включаются в набор. В общем случае шаблоны могут задаваться с использованием следующих метасимволов:

- `*` **– соответствует всем цепочкам литер, включая пустую;**
- `?` **– соответствует всем одиночным литерам;**
- `[...]` **– соответствует любой литере, заключенной в скобки. Пара литер, разделенных знаком минус, задает диапазон литер.**

Так, например, шаблону *.c **удовлетворяют все файлы текущей директории, чьи имена заканчиваются на** .c. **Шаблону** [a-d]* **удовлетворяют все файлы текущей директории, чьи имена начинаются с букв** a, b, c, d. **Существует лишь ограничение на использование метасимвола** * **в начале имени файла, например, в случае шаблона** *c. **Для таких шаблонов имена файлов, начинающиеся с символа точка, считаются не удовлетворяющими шаблону.**

Для удаления файлов или директорий применяется команда rm (сокращение от remove). Если вы хотите удалить один или несколько регулярных файлов, то простейший вид команды rm будет выглядеть следующим образом:

```
rm файл1 файл2 ... файлN
```

где `файл1, файл2, ... файлN` – полные или относительные имена регулярных файлов, которые требуется удалить. Вместо имен файлов могут использоваться их шаблоны. Если вы хотите удалить одну или несколько директорий вместе с их содержимым (рекурсивное удаление), то к команде добавляется опция `-r`:

```
rm -r дир1 дир2 ... дирN
```

где `дир1, дир2, ... дирN` – полные или относительные имена директорий, которые нужно удалить. Вместо непосредственно имен директорий также могут использоваться их шаблоны. У команды `rm` есть еще набор полезных опций, которые описаны в UNIX Manual. На самом деле процесс удаления файлов не так прост, как кажется на первый взгляд. Более подробно он будет рассмотрен нами на семинарах 11–12, когда мы будем обсуждать операции над файлами в операционной системе UNIX.

Команда mv

Синтаксис команды

```
mv имя_источника имя_назначения
mv имя1 имя2 ... имяN дир_назначения
```

Описание команды

Настоящее описание не является полным описанием команды `mv`, а служит кратким введением в ее использование. Для получения полного описания команды обращайтесь к UNIX Manual.

Команда `mv` в форме

```
mv имя_источника имя_назначения
```

служит для переименования или перемещения одного файла (неважно, регулярного или директории) с именем `имя_источника` в файл с именем `имя_назначения`. При этом перед выполнением команды файла с именем `имя_назначения` существовать не должно.

Команда `mv` в форме

```
mv имя1 имя2 ... имяN дир_назначения
```

служит для перемещения файла или файлов (неважно, регулярных файлов или директорий) с именами `имя1, имя2, ... имяN` в уже существующую директорию с именем `дир_назначения` под собственными именами. Вместо имен перемещаемых файлов могут использоваться их шаблоны.

Командой удаления файлов и директорий следует пользоваться с осторожностью. Удаленную информацию восстановить невозможно. Если вы системный администратор и ваша текущая директория – это корневая директория, пожалуйста, не выполняйте команду `rm -r *`!

Для переименования файла или его перемещения в другой каталог применяется команда `mv` (сокращение от move). Для задания имен перемещаемых файлов в ней тоже можно использовать их шаблоны.

История редактирования файлов – ed, vi

Полученные знания уже позволяют нам достаточно свободно оперировать файлами. Но что нам делать, если потребуется изменить содержимое файла, отредактировать его?

Когда появились первые варианты операционной системы UNIX, устройства ввода и отображения информации существенно отличались от существующих сегодня. На клавиатурах присутствовали только алфавитно-цифровые клавиши (не было даже клавиш курсоров), а дисплеи не предполагали экранного редактирования. Поэтому первый редактор операционной системы UNIX – редактор `ed` – требовал от пользователя строгого указания того, что и как будет редактироваться с помощью специальных команд. Так, например, для замены первого сочетания символов «`ra`» на «`ru`» в одиннадцатой строке редактируемого файла потребовалось бы ввести команду

```
11 s/ra/ru
```

Редактор `ed`*, по существу, являлся построчечным редактором. Впоследствии появился экранный редактор – `vi`**, однако и он требовал строгого указания того, что и как в текущей позиции на экране мы должны сделать, или каким образом изменить текущую позицию, с помощью специальных команд, соответствующих алфавитно-цифровым клавишам. Эти редакторы могут показаться нам сейчас анахронизмами, но они до сих пор входят в состав всех вариантов UNIX и иногда (например, при работе с удаленной машиной по медленному каналу связи) являются единственным средством, позволяющим удаленно редактировать файл.

---

* Описание редактора `ed` можно найти, например, в [Баурн, 1986]. В электронном виде описание есть в документе http://cs.mipt.ru/docs/comp/rus/os/unix/user_guide/unixuser/gl6_1.htm.

** Описание редактора `vi` а также можно найти в [Баурн, 1986]. В электронном виде описание есть в документе http://cs.mipt.ru/docs/comp/rus/os/unix/user_guide/unixuser/gl7_1.htm.

Система Midnight Commander – mc. Встроенный mc редактор и редактор joe

Наверное, вы уже убедились в том, что работа в UNIX исключительно на уровне командного интерпретатора и встроенных редакторов далека от уже привычных для нас удобств. Но не все так плохо. Существуют разнообразные пакеты, облегчающие задачу пользователя в UNIX. К таким пакетам следует отнести Midnight Commander – аналог программ Norton Commander для DOS и FAR для Windows 9x и NT – со своим встроенным редактором, запускаемый командой `mc`, и экранный редактор `joe`. Информацию о них можно найти в UNIX Manual. Большими возможностями обладают многофункциональные текстовые редакторы, например, `emacs`*.

Войдите в `mc` и попробуйте перемещаться по директориям, создавать и редактировать файлы.

Пользователь и группа. Команды chown и chgrp. Права доступа к файлу

Как уже говорилось, для входа в операционную систему UNIX каждый пользователь должен быть зарегистрирован в ней под определенным именем. Вычислительные системы не умеют оперировать именами, поэтому каждому имени пользователя в системе соответствует некоторое числовое значение – его идентификатор – `UID` (**U**ser **ID**entificator).

Все пользователи в системе делятся на группы. Например, студенты одной учебной группы могут составлять отдельную группу пользователей. Группы пользователей также получают свои имена и соответствующие идентификационные номера – `GID` (**G**roup **ID**entificator). В одних версиях UNIX каждый пользователь может входить только в одну группу, в других – в несколько групп.

Команда chown

Синтаксис команды

```
chown owner файл1 файл2 ... файлN
```

Описание команды

Команда `chown` **предназначена для изменения собственника (хозяина) файлов. Настоящее описание не является полным описанием команды, а адаптировано применительно к данному**

* В электронном виде описание редактора emacs см. в документе http://cs.mipt.ru/docs/comp/rus/os/unix/user_guide/emacs/index.html.

курсу. Для получения полного описания обращайтесь к UNIX Manual. Нового собственника файла могут назначить только предыдущий собственник файла или системный администратор.

Параметр owner задает нового собственника файла в символьном виде, как его username, или в числовом виде, как его UID.

Параметры файл1, файл2, ... файлN – это имена файлов, для которых производится изменение собственника. Вместо имен могут использоваться их шаблоны.

Для каждого файла, созданного в файловой системе, запоминаются имена его хозяина и группы хозяев. Заметим, что группа хозяев не обязательно должна быть группой, в которую входит хозяин. Упрощенно можно считать, что в операционной системе Linux при создании файла его хозяином становится пользователь, создавший файл, а его группой хозяев – группа, к которой этот пользователь принадлежит. Впоследствии хозяин файла или системный администратор могут передать его в собственность другому пользователю или изменить его группу хозяев с помощью команд chown и chgrp, описание которых можно найти в UNIX Manual.

Команда chgrp

Синтаксис команды

```
chgrp group файл1 файл2 ... файлN
```

Описание команды

Команда chgrp предназначена для изменения группы собственников (хозяев) файлов. Настоящее описание не является полным описанием команды, а адаптировано применительно к данному курсу. Для получения полного описания обращайтесь к UNIX Manual. Новую группу собственников файла могут назначить только собственник файла или системный администратор.

Параметр group задает новую группу собственников файла в символьном виде, как имя группы, или в числовом виде, как ее GID.

Параметры файл1, файл2, ... файлN – это имена файлов, для которых производится изменение группы собственников. Вместо имен могут использоваться их шаблоны.

Как мы видим, для каждого файла выделяется три категории пользователей:

- Пользователь, являющийся хозяином файла.
- Пользователи, относящиеся к группе хозяев файла.
- Все остальные пользователи.

Для каждой из этих категорий хозяин файла может определить различные права доступа к файлу. Различают три вида прав доступа: право на чтение файла – r (от слова read), право на модификацию файла – w (от слова write) и право на исполнение файла — x (от слова execute). Для регуляр-

ных файлов смысл этих прав совпадает с указанным выше. Для директорий он несколько иной. Право чтения для каталогов позволяет читать имена файлов, находящихся в этом каталоге (и только имена). Поскольку «исполнять» директорию бессмысленно (как, впрочем, и неисполняемый регулярный файл), право доступа на исполнение для директорий меняет смысл: наличие этого права позволяет получить дополнительную информацию о файлах, входящих в каталог (их размер, кто их хозяин, дата создания и т. д.). Без этого права вы не сможете ни читать содержимое файлов, лежащих в директории, ни модифицировать их, ни исполнять. Право на исполнение также требуется для директории, чтобы сделать ее текущей, а также для всех директорий на пути к ней. Право записи для директории позволяет изменять ее содержимое: создавать и удалять в ней файлы, переименовывать их. Отметим, что для удаления файла достаточно иметь права записи и исполнения для директории, в которую входит данный файл, независимо от прав доступа к самому файлу.

Команда ls с опциями -al. Использование команд chmod и umask

Получить подробную информацию о файлах в некоторой директории, включая имена хозяина, группы хозяев и права доступа, можно с помощью уже известной нам команды `ls` с опциями `-al`. В выдаче этой команды третья колонка слева содержит имена пользователей хозяев файлов, а четвертая колонка слева – имена групп хозяев файла. Самая левая колонка содержит типы файлов и права доступа к ним. Тип файла определяет первый символ в наборе символов. Если это символ 'd', то тип файла – директория, если там стоит символ '-', то это – регулярный файл. Следующие три символа определяют права доступа для хозяина файла, следующие три – для пользователей, входящих в группу хозяев файла, и последние три – для всех остальных пользователей. Наличие символа (`r`, `w` или `x`), соответствующего праву, для некоторой категории пользователей означает, что данная категория пользователей обладает этим правом.

Вызовите команду `ls -al` для своей домашней директории и проанализируйте ее выдачу.

Команда chmod

Синтаксис команды

```
chmod [who] { + | - | = } [perm] файл1 файл2 ... файлN
```

Описание команды

Команда chmod предназначена для изменения прав доступа к одному или нескольким файлам. Настоящее описание не является полным описанием команды, а адаптировано применительно к данному курсу. Для получения полного описания обращайтесь к UNIX Manual. Права доступа к файлу могут менять только собственник (хозяин) файла или системный администратор.

Параметр who определяет, для каких категорий пользователей устанавливаются права доступа. Он может представлять собой один или несколько символов:

a – установка прав доступа для всех категорий пользователей. Если параметр who не задан, то по умолчанию применяется a. При определении прав доступа с этим значением заданные права устанавливаются с учетом значения маски создания файлов;

u – установка прав доступа для собственника файла;

g – установка прав доступа для пользователей, входящих в группу собственников файла;

o – установка прав доступа для всех остальных пользователей.

Операция, выполняемая над правами доступа для заданной категории пользователей, определяется одним из следующих символов:

+ – добавление прав доступа;

- – отмена прав доступа;

= – замена прав доступа, т.е. отмена всех существовавших и добавление перечисленных. Если параметр perm не определен, то все существовавшие права доступа отменяются.

Параметр perm определяет права доступа, которые будут добавлены, отменены или установлены взамен соответствующей командой. Он представляет собой комбинацию следующих символов или один из них:

r – право на чтение;

w – право на модификацию;

x – право на исполнение.

Параметры файл1, файл2, ... файлN – это имена файлов, для которых производится изменение прав доступа. Вместо имен могут использоваться их шаблоны.

Хозяин файла может изменять права доступа к нему, пользуясь командой chmod.

Создайте новый файл и посмотрите на права доступа к нему, установленные системой при его создании. Чем руководствуется операционная система при назначении этих прав? Она использует для этого маску создания файлов для программы, которая файл создает. Изначально для программы-оболочки она имеет некоторое значение по умолчанию.

Маска создания файлов текущего процесса

Маска создания файлов текущего процесса (umask) используется системными вызовами open() и mknod() при установке начальных прав доступа для вновь создаваемых файлов или FIFO. Младшие 9 бит маски создания файлов соответствуют правам доступа пользователя, соз-

дающего файл, группы, к которой он принадлежит, и всех остальных пользователей так, как записано ниже с применением восьмеричных значений:

0400 – право чтения для пользователя, создавшего файл;

0200 – право записи для пользователя, создавшего файл;

0100 – право исполнения для пользователя, создавшего файл;

0040 – право чтения для группы пользователя, создавшего файл;

0020 – право записи для группы пользователя, создавшего файл;

0010 – право исполнения для группы пользователя, создавшего файл;

0004 – право чтения для всех остальных пользователей;

0002 – право записи для всех остальных пользователей;

0001 – право исполнения для всех остальных пользователей.

Установление значения какого-либо бита равным 1 запрещает инициализацию соответствующего права доступа для вновь создаваемого файла. Значение маски создания файлов может изменяться с помощью системного вызова umask() или команды umask. Маска создания файлов наследуется процессом-ребенком при порождении нового процесса системным вызовом fork() и входит в состав неизменяемой части системного контекста процесса при системном вызове exec(). В результате этого наследования изменение маски с помощью команды umask окажет влияние на атрибуты доступа к вновь создаваемым файлам для всех процессов, порожденных далее командной оболочкой.

Изменить текущее значение маски для программы-оболочки или посмотреть его можно с помощью команды umask.

Команда umask

Синтаксис команды

```
umask [value]
```

Описание команды

Команда umask предназначена для изменения маски создания файлов командной оболочки или просмотра ее текущего значения. При отсутствии параметра команда выдает значение установленной маски создания файлов в восьмеричном виде. Для установления нового значения оно задается как параметр value в восьмеричном виде.

Если вы хотите изменить его для Midnight Commander, необходимо выйти из mc, выполнить команду umask и запустить mc снова. Маска создания файлов не сохраняется между сеансами работы в системе. При новом входе в систему значение маски снова будет установлено по умолчанию.

Системные вызовы getuid и getgid

Узнать идентификатор пользователя, запустившего программу на исполнение, – UID и идентификатор группы, к которой он относится, – GID можно с помощью системных вызовов getuid() и getgid(), применив их внутри этой программы.

Системные вызовы getuid() и getgid()

Прототипы системных вызовов

```
#include <sys/types.h>
#include <unistd.h>
uid_t getuid(void);
gid_t getgid(void);
```

Описание системных вызовов

Системный вызов getuid возвращает идентификатор пользователя для текущего процесса.

Системный вызов getgid возвращает идентификатор группы пользователя для текущего процесса.

Типы данных uid_t и gid_t являются синонимами для одного из целочисленных типов языка C.

Компиляция программ на языке C в UNIX и запуск их на счет

Теперь мы готовы к тому, чтобы написать первую программу в нашем курсе. Осталось только научиться компилировать программы на языке C и запускать их на счет. Для компиляции программ в Linux мы будем применять компилятор gcc.

Для того чтобы он нормально работал, необходимо, чтобы исходные файлы, содержащие текст программы, имели имена, заканчивающиеся на .c.

В простейшем случае откомпилировать программу можно, запуская компилятор командой

```
gcc имя_исходного_файла
```

Если программа была написана без ошибок, то компилятор создаст исполняемый файл с именем a.out. Изменить имя создаваемого исполняемого файла можно, задав его с помощью опции `-o`.

```
gcc имя_исходного_файла -o имя_исполняемого_файла
```

Компилятор `gcc` имеет несколько сотен возможных опций. Получить информацию о них вы можете в UNIX Manual.

«Узнайте у своего системного администратора», как называется компилятор с языка С для вашей операционной системы и какие опции он имеет. Обычно во всех версиях UNIX имеется компилятор с именем `cc`, поддерживающий опцию `-o`.

Запустить программу на исполнение можно, набрав имя исполняемого файла и нажав клавишу <Enter>.

Написание, компиляция и запуск программы с использованием системных вызовов getuid() и getgid()

Напишите, откомпилируйте и запустите программу, которая печатала бы идентификатор пользователя, запустившего программу, и идентификатор его группы.

Семинары 3–4. Процессы в операционной системе UNIX

Понятие процесса в UNIX, его контекст. Идентификация процесса. Состояния процесса. Краткая диаграмма состояний. Иерархия процессов. Системные вызовы `getpid()`, `getppid()`. Создание процесса в UNIX. Системный вызов `fork()`. Завершение процесса. Функция `exit()`. Параметры функции `main()` в языке С. Переменные среды и аргументы командной строки. Изменение пользовательского контекста процесса. Семейство функций для системного вызова `exec()`.

Ключевые слова: пользовательский контекст процесса, контекст ядра, идентификатор процесса, PID, идентификатор родительского процесса, PPID, исполнение в режиме пользователя, исполнение в режиме ядра, системные вызовы `getpid`, `getppid`, `fork`, `exec`, создание процесса, завершение процесса, функции `exit`, `execl`, `execv`, `execlp`, `execvp`, `execle`, `execve`, параметры функции `main`, аргументы командной строки, переменные среды.

Понятие процесса в UNIX. Его контекст

Все построение операционной системы UNIX основано на использовании концепции процессов, которая обсуждалась на лекции. Контекст процесса складывается из пользовательского контекста и контекста ядра, как изображено на рис. 3-4.1.

Под пользовательским контекстом процесса понимают код и данные, расположенные в адресном пространстве процесса. Все данные подразделяются на:

- инициализируемые неизменяемые данные (например, константы);
- инициализируемые изменяемые данные (все переменные, начальные значения которых присваиваются на этапе компиляции);
- неинициализируемые изменяемые данные (все статические переменные, которым не присвоены начальные значения на этапе компиляции);
- стек пользователя;
- данные, расположенные в динамически выделяемой памяти (например, с помощью стандартных библиотечных С функций `malloc()`, `calloc()`, `realloc()`).

Исполняемый код и инициализируемые данные составляют содержимое файла программы, который исполняется в контексте процесса. Пользовательский стек применяется при работе процесса в пользовательском режиме (user-mode).

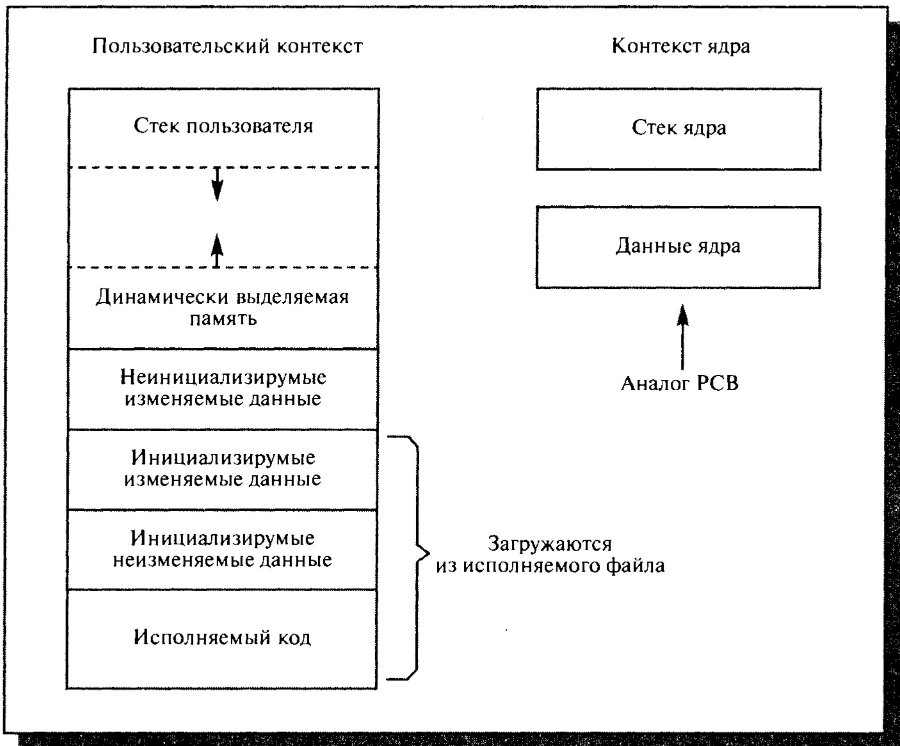


Рис. 3-4.1. Контекст процесса в UNIX

Под понятием «контекст ядра» объединяются системный контекст и регистровый контекст, рассмотренные на лекции. Мы будем выделять в контексте ядра стек ядра, который используется при работе процесса в режиме ядра (kernel mode), и данные ядра, хранящиеся в структурах, являющихся аналогом блока управления процессом — PCB. Состав данных ядра будет уточняться на последующих семинарах. На этом занятии нам достаточно знать, что в данные ядра входят: идентификатор пользователя — `UID`, групповой идентификатор пользователя — `GID`, идентификатор процесса — `PID`, идентификатор родительского процесса — `PPID`.

Идентификация процесса

Каждый процесс в операционной системе получает уникальный идентификационный номер – `PID` (**P**rocess **ID**entificator). При создании нового процесса операционная система пытается присвоить ему свободный номер больший, чем у процесса, созданного перед ним. Если таких

свободных номеров не оказывается (например, мы достигли максимально возможного номера для процесса), то операционная система выбирает минимальный номер из всех свободных номеров. В операционной системе Linux присвоение идентификационных номеров процессов начинается с номера 0, который получает процесс *kernel* при старте операционной системы. Этот номер впоследствии не может быть присвоен никакому другому процессу. Максимально возможное значение для номера процесса в Linux на базе 32-разрядных процессоров Intel составляет $2^{31} - 1$.

Состояния процесса. Краткая диаграмма состояний

Модель состояний процессов в операционной системе UNIX представляет собой детализацию модели состояний, принятой в лекционном курсе. Краткая диаграмма состояний процессов в операционной системе UNIX изображена на рис. 3-4.2.

Как мы видим, состояние процесса ***исполнение*** расщепилось на два состояния: ***исполнение в режиме ядра*** и ***исполнение в режиме пользователя***. В состоянии ***исполнение в режиме пользователя*** процесс выполняет прикладные инструкции пользователя. В состоянии ***исполнение в режиме ядра*** выполняются инструкции ядра операционной системы в контексте текущего процесса (например, при обработке системного вызова или прерыва-

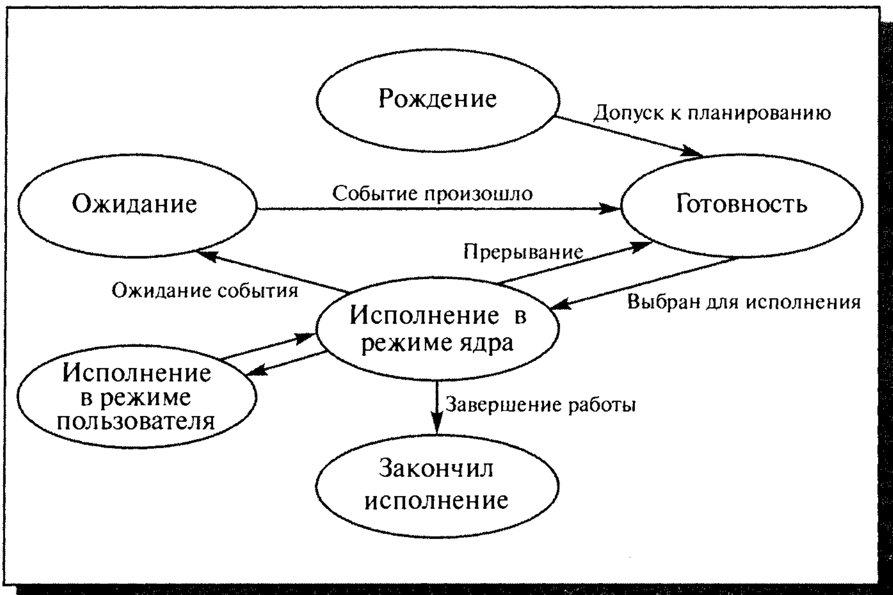


Рис. 3-4.2. Сокращенная диаграмма состояний процесса в UNIX

ния). Из состояния ***исполнение в режиме пользователя*** процесс не может непосредственно перейти в состояния ***ожидание***, ***готовность*** и ***закончил исполнение***. Такие переходы возможны только через промежуточное состояние ***исполнение в режиме ядра***. Также запрещен прямой переход из состояния ***готовность*** в состояние ***исполнение в режиме пользователя***.

Приведенная выше диаграмма состояний процессов в UNIX не является полной. Она показывает только состояния, для понимания которых достаточно уже полученных знаний. Пожалуй, наиболее полную диаграмму состояний процессов в операционной системе UNIX можно найти в книге [Bach, 1986] (рис. 6.1).

Иерархия процессов

В операционной системе UNIX все процессы, кроме одного, создающегося при старте операционной системы, могут быть порождены только какими-либо другими процессами. В качестве прародителя всех остальных процессов в подобных UNIX-системах могут выступать процессы с номерами 1 или 0. В операционной системе Linux таким родоначальником, существующим только при загрузке системы, является процесс *kernel* с идентификатором 0.

Таким образом, все процессы в UNIX связаны отношениями процесс-родитель – процесс-ребенок и образуют генеалогическое дерево процессов. Для сохранения целостности генеалогического дерева в ситуациях, когда процесс-родитель завершает свою работу до завершения выполнения процесса-ребенка, идентификатор родительского процесса в данных ядра процесса-ребенка (`PPID` – **P**arent **P**rocess **ID**entificator) изменяет свое значение на значение 1, соответствующее идентификатору процесса *init*, время жизни которого определяет время функционирования операционной системы. Тем самым процесс *init* как бы усыновляет осиротевшие процессы. Наверное, логичнее было бы заменять `PPID` не на значение 1, а на значение идентификатора ближайшего существующего процесса-прародителя умершего процесса-родителя, но в UNIX почему-то такая схема реализована не была.

Системные вызовы getppid() и getpid()

Данные ядра, находящиеся в контексте ядра процесса, не могут быть прочитаны процессом непосредственно. Для получения информации о них процесс должен совершить соответствующий системный вызов. Значение идентификатора текущего процесса может быть получено с помощью системного вызова `getpid()`, а значение идентификатора родительского процесса для текущего процесса – с помощью системного вызова `getppid()`. Прототипы этих системных вызовов и соответствую-

щие типы данных описаны в системных файлах <sys/types.h> и <unistd.h>. Системные вызовы не имеют параметров и возвращают идентификатор текущего процесса и идентификатор родительского процесса соответственно.

Системные вызовы getpid() и getppid()

Прототипы системных вызовов

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Описание системных вызовов

Системный вызов getpid **возвращает идентификатор текущего процесса.**

Системный вызов getppid **возвращает идентификатор процесса-родителя для текущего процесса.**

Тип данных pid_t **является синонимом для одного из целочисленных типов языка С.**

Написание программы с использованием getpid() и getppid()

В качестве примера использования системных вызовов getpid() и getppid() самостоятельно напишите программу, печатающую значения PID и PPID для текущего процесса. Запустите ее несколько раз подряд. Посмотрите, как меняется идентификатор текущего процесса. Объясните наблюдаемые изменения.

Создание процесса в UNIX. Системный вызов fork()

В операционной системе UNIX новый процесс может быть порожден единственным способом – с помощью системного вызова fork(). При этом вновь созданный процесс будет являться практически полной копией родительского процесса. У порожденного процесса по сравнению с родительским процессом (на уровне уже полученных знаний) изменяются значения следующих параметров:

- идентификатор процесса – PID;
- идентификатор родительского процесса – PPID.

Дополнительно может измениться поведение порожденного процесса по отношению к некоторым сигналам, о чем подробнее будет рас-

сказано на семинарах 13–14, когда мы будем говорить о сигналах в операционной системе UNIX.

Системный вызов для порождения нового процесса

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Описание системного вызова

Системный вызов `fork` служит для создания нового процесса в операционной системе UNIX. Процесс, который инициировал системный вызов `fork`, принято называть родительским процессом (parent process). Вновь порожденный процесс принято называть процессом-ребенком (child process). Процесс-ребенок является почти полной копией родительского процесса. У порожденного процесса по сравнению с родительским изменяются значения следующих параметров:

- идентификатор процесса;
- идентификатор родительского процесса;
- время, оставшееся до получения сигнала `SIGALRM`;
- сигналы, ожидавшие доставки родительскому процессу, не будут доставляться порожденному процессу.

При однократном системном вызове возврат из него может произойти дважды: один раз в родительском процессе, а второй раз в порожденном процессе. Если создание нового процесса произошло успешно, то в порожденном процессе системный вызов вернет значение 0, а в родительском процессе – положительное значение, равное идентификатору процесса-ребенка. Если создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс отрицательное значение.

Системный вызов `fork` является единственным способом породить новый процесс после инициализации операционной системы UNIX.

В процессе выполнения системного вызова `fork()` порождается копия родительского процесса и возвращение из системного вызова будет происходить уже как в родительском, так и в порожденном процессах. Этот системный вызов является единственным, который вызывается один раз, а при успешной работе возвращается два раза (один раз в процессе-родителе и один раз в процессе-ребенке)! После выхода из системного вызова оба процесса продолжают выполнение регулярного пользовательского кода, следующего за системным вызовом.

Прогон программы с fork() с одинаковой работой родителя и ребенка

Для иллюстрации сказанного давайте рассмотрим следующую программу

```
/* Программа 03-1.c - пример создания нового процесса
с одинаковой работой процессов ребенка и родителя */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
  pid_t pid, ppid;
  int a = 0;
  (void)fork();
  /* При успешном создании нового процесса с этого
  места псевдопараллельно начинают работать два
  процесса: старый и новый */
  /* Перед выполнением следующего выражения значение
  переменной a в обоих процессах равно 0 */
  a = a+1;
  /* Узнаем идентификаторы текущего и родительского
  процесса (в каждом из процессов !!!) */
  pid = getpid();
  ppid = getppid();
  /* Печатаем значения PID, PPID и вычисленное
  значение переменной a (в каждом из процессов !!!) */
  printf("My pid = %d, my ppid = %d, result = %d\n",
    (int)pid, (int)ppid, a);
  return 0;
}
```

Наберите эту программу, откомпилируйте ее и запустите на исполнение (лучше всего это делать не из оболочки mc, так как она не очень корректно сбрасывает буферы ввода-вывода). Проанализируйте полученный результат.

Системный вызов fork() (продолжение)

Для того чтобы после возвращения из системного вызова `fork()` процессы могли определить, кто из них является ребенком, а кто родителем,

и, соответственно, по-разному организовать свое поведение, системный вызов возвращает в них разные значения. При успешном создании нового процесса в процесс-родитель возвращается положительное значение, равное идентификатору процесса-ребенка. В процесс-ребенок же возвращается значение 0. Если по какой-либо причине создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс значение −1. Таким образом, общая схема организации различной работы процесса-ребенка и процесса-родителя выглядит так:

```
pid = fork();
if(pid == -1){
    ...
    /* ошибка */
    ...
} else if (pid == 0){
    ...
    /* ребенок */
    ...
} else {
    ...
    /* родитель */
    ...
}
```

Написание, компиляция и запуск программы с использованием вызова fork() с разным поведением процессов ребенка и родителя

Измените предыдущую программу с `fork()` так, чтобы родитель и ребенок совершали разные действия (какие – не важно).

Завершение процесса. Функция exit()

Существует два способа корректного завершения процесса в программах, написанных на языке C. Первый способ мы использовали до сих пор: процесс корректно завершался по достижении конца функции `main()` или при выполнении оператора `return` в функции `main()`, второй способ применяется при необходимости завершить процесс в каком-либо другом месте программы. Для этого используется функция `exit()` из стандартной библиотеки функций для языка C. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-

вывода с закрытием соответствующих потоков, после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние ***закончил исполнение***.

Возврата из функции в текущий процесс не происходит и функция ничего не возвращает.

Значение параметра функции `exit()` — кода завершения процесса — передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. На самом деле при достижении конца функции main() также неявно вызывается эта функция со значением параметра 0.

Функция для нормального завершения процесса

Прототип функции

```
#include <stdlib.h>
void exit(int status);
```

Описание функции

Функция `exit` служит для нормального завершения процесса. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков (файлов, pipe, FIFO, сокетов), после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние закончил исполнение.

Возврата из функции в текущий процесс не происходит, и функция ничего не возвращает.

Значение параметра `status` – кода завершения процесса – передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. При этом используются только младшие 8 бит параметра, так что для кода завершения допустимы значения от 0 до 255. По соглашению, код завершения 0 означает безошибочное завершение процесса.

Если процесс завершает свою работу раньше, чем его родитель, и родитель явно не указал, что он не хочет получать информацию о статусе завершения порожденного процесса (об этом будет рассказано подробнее на семинарах 13–14 при изучении сигналов), то завершившийся процесс не исчезает из системы окончательно, а остается в состоянии ***закончил исполнение*** либо до завершения процесса-родителя, либо до того момента, когда родитель получит эту информацию. Процессы, находящиеся в состоянии ***закончил исполнение***, в операционной системе UNIX принято называть процессами-зомби (zombie, defunct).

Параметры функции main() в языке С. Переменные среды и аргументы командной строки

У функции main() в языке программирования С существует три параметра, которые могут быть переданы ей операционной системой. Полный прототип функции main() выглядит следующим образом:

```
int main(int argc, char *argv[], char *envp[]);
```

Первые два параметра при запуске программы на исполнение командной строкой позволяют узнать полное содержание командной строки. Вся командная строка рассматривается как набор слов, разделенных пробелами. Через параметр argc передается количество слов в командной строке, которой была запущена программа. Параметр argv является массивом указателей на отдельные слова. Так, например, если программа была запущена командой

```
a.out 12 abcd
```

то значение параметра argc будет равно 3, argv[0] будет указывать на имя программы — первое слово — "a.out", argv[1] — на слово "12", argv[2] — на слово "abcd". Так как имя программы всегда присутствует на первом месте в командной строке, то argc всегда больше 0, а argv[0] всегда указывает на имя запущенной программы.

Анализируя в программе содержимое командной строки, мы можем предусмотреть ее различное поведение в зависимости от слов, следующих за именем программы. Таким образом, не внося изменений в текст программы, мы можем заставить ее работать по-разному от запуска к запуску. Например, компилятор gcc, вызванный командой gcc 1.c будет генерировать исполняемый файл с именем a.out, а при вызове командой gcc 1.c -o 1.exe – файл с именем 1.exe.

Третий параметр – envp – является массивом указателей на параметры окружающей среды процесса. Начальные параметры окружающей среды процесса задаются в специальных конфигурационных файлах для каждого пользователя и устанавливаются при входе пользователя в систему. В дальнейшем они могут быть изменены с помощью специальных команд операционной системы UNIX. Каждый параметр имеет вид: переменная=строка. Такие переменные используются для изменения долгосрочного поведения процессов, в отличие от аргументов командной строки. Например, задание параметра TERM=vt100 может говорить процессам, осуществляющим вывод на экран дисплея, что работать им придется с терминалом vt100. Меняя значение переменной среды TERM, например

на TERM=console, мы сообщаем таким процессам, что они должны изменить свое поведение и осуществлять вывод для системной консоли.

Размер массива аргументов командной строки в функции main() мы получали в качестве ее параметра. Так как для массива ссылок на параметры окружающей среды такого параметра нет, то его размер определяется другим способом. Последний элемент этого массива содержит указатель NULL.

Написание, компиляция и запуск программы, распечатывающей аргументы командной строки и параметры среды

В качестве примера самостоятельно напишите программу, распечатывающую значения аргументов командной строки и параметров окружающей среды для текущего процесса.

Изменение пользовательского контекста процесса. Семейство функций для системного вызова exec()

Для изменения пользовательского контекста процесса применяется системный вызов exec(), который пользователь не может вызвать непосредственно. Вызов exec() заменяет пользовательский контекст текущего процесса на содержимое некоторого исполняемого файла и устанавливает начальные значения регистров процессора (в том числе устанавливает программный счетчик на начало загружаемой программы). Этот вызов требует для своей работы задания имени исполняемого файла, аргументов команд-

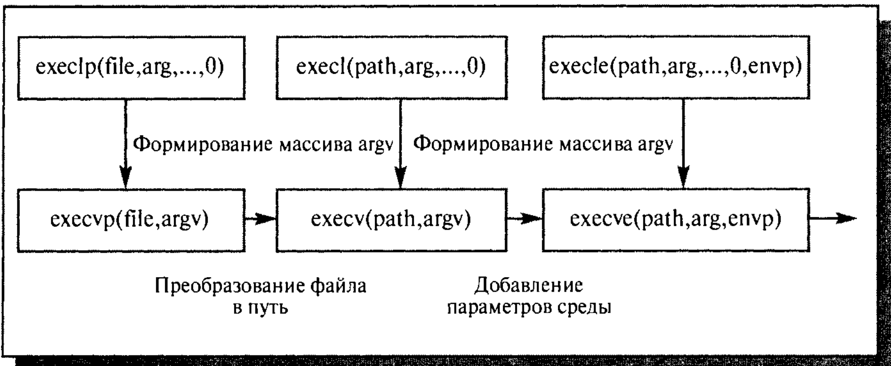


Рис. 3-4.3. Взаимосвязь различных функций для выполнения системного вызова exec()

ной строки и параметров окружающей среды. Для осуществления вызова программист может воспользоваться одной из шести функций: `execlp()`, `execvp()`, `execl()`, `execv()`, `execle()`, `execve()`, отличающихся друг от друга представлением параметров, необходимых для работы системного вызова `exec()`. Взаимосвязь указанных выше функций изображена на рисунке 3-4.3.

Функции изменения пользовательского контекста процесса

Прототипы функций

```
#include <unistd.h>
int execlp(const char *file, const char *arg0,
   ... const char *argN,(char *)NULL)
int execvp(const char *file, char *argv[])
int execl(const char *path, const char *arg0,
   ... const char *argN,(char *)NULL)
int execv(const char *path, char *argv[])
int execle(const char *path, const char *arg0,
   ... const char *argN,(char *)NULL, char * envp[])
int execve(const char *path, char *argv[], char *envp[])
```

Описание функций

Для загрузки новой программы в системный контекст текущего процесса используется семейство взаимосвязанных функций, отличающихся друг от друга формой представления параметров.

Аргумент `file` является указателем на имя файла, который должен быть загружен. Аргумент `path` – это указатель на полный путь к файлу, который должен быть загружен.

Аргументы `arg0, ..., argN` представляют собой указатели на аргументы командной строки. Заметим, что аргумент `arg0` должен указывать на имя загружаемого файла. Аргумент `argv` представляет собой массив из указателей на аргументы командной строки. Начальный элемент массива должен указывать на имя загружаемой программы, а заканчиваться массив должен элементом, содержащим указатель `NULL`.

Аргумент `envp` является массивом указателей на параметры окружающей среды, заданные в виде строк «переменная=строка». Последний элемент этого массива должен содержать указатель `NULL`.

Поскольку вызов функции не изменяет системный контекст текущего процесса, загруженная программа унаследует от загрузившего ее процесса следующие атрибуты:

- идентификатор процесса;
- идентификатор родительского процесса;
- групповой идентификатор процесса;
- идентификатор сеанса;

- время, оставшееся до возникновения сигнала SIGALRM;
- текущую рабочую директорию;
- маску создания файлов;
- идентификатор пользователя;
- групповой идентификатор пользователя;
- явное игнорирование сигналов;
- таблицу открытых файлов (если для файлового дескриптора не устанавливался признак «закрыть файл при выполнении exec()»).

В случае успешного выполнения возврата из функций в программу, осуществившую вызов, не происходит, а управление передается загруженной программе. В случае неудачного выполнения в программу, инициировавшую вызов, возвращается отрицательное значение.

Поскольку системный контекст процесса при вызове exec() остается практически неизменным, большинство атрибутов процесса, доступных пользователю через системные вызовы (PID, UID, GID, PPID и другие, смысл которых станет понятен по мере углубления наших знаний на дальнейших занятиях), после запуска новой программы также не изменяется.

Важно понимать разницу между системными вызовами fork() и exec(). Системный вызов fork() создает новый процесс, у которого пользовательский контекст совпадает с пользовательским контекстом процесса-родителя. Системный вызов exec() изменяет пользовательский контекст текущего процесса, не создавая новый процесс.

Прогон программы с использованием системного вызова exec()

Для иллюстрации использования системного вызова exec() давайте рассмотрим следующую программу:

```
/* Программа 03-2.c, изменяющая пользовательский
контекст процесса (запускающая другую программу) */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[], char *envp[]){
    /* Мы будем запускать команду cat с аргументом
    командной строки 03-2.c без изменения параметров
    среды, т.е. фактически выполнять команду
    "cat 03-2.c", которая должна выдать содержимое
    данного файла на экран. Для функции execle в
```

```
    качестве имени программы мы указываем ее полное
    имя с путем от корневой директории –/bin/cat.
    Первое слово в командной строке у нас должно
    совпадать с именем запускаемой программы. Второе
    слово в командной строке - это имя файла,
    содержимое которого мы хотим распечатать. */
    (void) execle("/bin/cat", "/bin/cat", "03-2.c",
       0, envp);
    /* Сюда попадаем только при возникновении ошибки */
    printf("Error on program start\n");
    exit(-1);
    return 0;  /* Никогда не выполняется, нужен для
                  того, чтобы компилятор не выдавал
                  warning */
}
```

Откомпилируйте ее и запустите на исполнение. Поскольку при нормальной работе будет распечатываться содержимое файла с именем 03-2.c, такой файл при запуске должен присутствовать в текущей директории (проще всего записать исходный текст программы под этим именем). Проанализируйте результат.

Написание, компиляция и запуск программы для изменения пользовательского контекста в порожденном процессе

Для закрепления полученных знаний модифицируйте программу, созданную при выполнении задания раздела «Написание, компиляция и запуск программы с использованием вызова `fork()` с разным поведением процессов ребенка и родителя» так, чтобы порожденный процесс запускал на исполнение новую (любую) программу.

Семинар 5. Организация взаимодействия процессов через pipe и FIFO в UNIX

Понятие потока ввода-вывода. Представление о работе с файлами через системные вызовы и стандартную библиотеку ввода-вывода. Понятие файлового дескриптора. Открытие файла. Системный вызов `open()`. Системные вызовы `close()`, `read()`, `write()`. Понятие pipe. Системный вызов `pipe()`. Организация связи через pipe между процессом-родителем и процессом-потомком. Наследование файловых дескрипторов при вызовах `fork()` и `exec()`. Особенности поведения вызовов `read()` и `write()` для pip'а. Понятие FIFO. Использование системного вызова `mknod()` для создания FIFO. Функция `mkfifo()`. Особенности поведения вызова `open()` при открытии FIFO.

Ключевые слова: поток ввода-вывода, файловый дескриптор, таблица открытых файлов процесса, стандартный поток ввода, стандартный поток вывода, стандартный поток вывода для ошибок, открытие файла, закрытие файла, pipe, FIFO, файл типа FIFO, системные вызовы `open`, `read`, `write`, `close`, `pipe`, `mknod`, `mkfifo`.

Понятие о потоке ввода-вывода

Как уже упоминалось в лекции 4, среди всех категорий средств коммуникации наиболее употребительными являются каналы связи, обеспечивающие достаточно безопасное и достаточно информативное взаимодействие процессов.

Существует две модели передачи данных по каналам связи – поток ввода-вывода и сообщения. Из них более простой является потоковая модель, в которой операции передачи/приема информации вообще не интересуются содержимым того, что передается или принимается. Вся информация в канале связи рассматривается как непрерывный поток байт, не обладающий никакой внутренней структурой. Изучению механизмов, обеспечивающих потоковую передачу данных в операционной системе UNIX, и будет посвящен этот семинар.

Понятие о работе с файлами через системные вызовы и стандартную библиотеку ввода-вывода для языка C

Потоковая передача информации может осуществляться не только между процессами, но и между процессом и устройством ввода-вывода,

например между процессом и диском, на котором данные представляются в виде файла. Поскольку понятие файла должно быть знакомо изучающим этот курс, а системные вызовы, использующиеся для потоковой работы с файлом, во многом соответствуют системным вызовам, применяемым для потокового общения процессов, мы начнем наше рассмотрение именно с механизма потокового обмена между процессом и файлом.

Как мы надеемся, из курса программирования на языке С вам известны функции работы с файлами из стандартной библиотеки ввода-вывода, такие как `fopen()`, `fread()`, `fwrite()`, `fprintf()`, `fscanf()`, `fgets()` и т. д. Эти функции входят как неотъемлемая часть в стандарт ANSI на язык С и позволяют программисту получать информацию из файла или записывать ее в файл при условии, что программист обладает определенными знаниями о содержимом передаваемых данных. Так, например, функция `fgets()` используется для ввода из файла последовательности символов, заканчивающейся символом '\n' – перевод каретки. Функция `fscanf()` производит ввод информации, соответствующей заданному формату, и т.д. С точки зрения потоковой модели операции, определяемые функциями стандартной библиотеки ввода-вывода, не являются потоковыми операциями, так как каждая из них требует наличия некоторой структуры передаваемых данных.

В операционной системе UNIX эти функции представляют собой надстройку – сервисный интерфейс – над системными вызовами, осуществляющими прямые потоковые операции обмена информацией между процессом и файлом и не требующими никаких знаний о том, что она содержит. Чуть позже мы кратко познакомимся с системными вызовами `open()`, `read()`, `write()` и `close()`, которые применяются для такого обмена, но сначала нам нужно ввести еще одно понятие – понятие *файлового дескриптора*.

Файловый дескриптор

В лекции 2 мы говорили, что информация о файлах, используемых процессом, входит в состав его системного контекста и хранится в его блоке управления – PCB. В операционной системе UNIX можно упрощенно полагать, что информация о файлах, с которыми процесс осуществляет операции потокового обмена, наряду с информацией о потоковых линиях связи, соединяющих процесс с другими процессами и устройствами ввода-вывода, хранится в некотором массиве, получившем название таблицы открытых файлов или таблицы файловых дескрипторов. Индекс элемента этого массива, соответствующий определенному потоку ввода-вывода, получил название файлового дескриптора для этого потока. Таким образом, файловый дескриптор представляет собой небольшое целое

неотрицательное число, которое для текущего процесса в данный момент времени однозначно определяет некоторый действующий канал ввода-вывода. Некоторые файловые дескрипторы на этапе старта любой программы ассоциируются со стандартными потоками ввода-вывода. Так, например, файловый дескриптор 0 соответствует стандартному потоку ввода, файловый дескриптор 1 – стандартному потоку вывода, файловый дескриптор 2 – стандартному потоку для вывода ошибок. В нормальном интерактивном режиме работы стандартный поток ввода связывает процесс с клавиатурой, а стандартные потоки вывода и вывода ошибок – с текущим терминалом.

Более детально строение структур данных, содержащих информацию о потоках ввода-вывода, ассоциированных с процессом, мы будем рассматривать позже, при изучении организации файловых систем в UNIX (семинары 11–12 и 13–14).

Открытие файла. Системный вызов open()

Файловый дескриптор используется в качестве параметра, описывающего поток ввода-вывода, для системных вызовов, выполняющих операции над этим потоком. Поэтому прежде чем совершать операции чтения данных из файла и записи их в файл, мы должны поместить информацию о файле в таблицу открытых файлов и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом `open()`.

Системный вызов open

Прототип системного вызова

```
#include <fcntl.h>
int open(char *path, int flags);
int open(char *path, int flags, int mode);
```

Описание системного вызова

Системный вызов `open` предназначен для выполнения операции открытия файла и, в случае ее удачного осуществления, возвращает файловый дескриптор открытого файла (небольшое неотрицательное целое число, которое используется в дальнейшем для других операций с этим файлом).

Параметр `path` является указателем на строку, содержащую полное или относительное имя файла.

Параметр `flags` может принимать одно из следующих трех значений:

O_RDONLY – если над файлом в дальнейшем будут совершаться только операции чтения;

O_WRONLY – если над файлом в дальнейшем будут осуществляться только операции записи;

O_RDWR – если над файлом будут осуществляться и операции чтения, и операции записи.

Каждое из этих значений может быть скомбинировано посредством операции «побитовое или (|)» с одним или несколькими флагами:

O_CREAT – если файла с указанным именем не существует, он должен быть создан;

O_EXCL – применяется совместно с флагом O_CREAT. При совместном их использовании и существовании файла с указанным именем, открытие файла не производится и констатируется ошибочная ситуация;

O_NDELAY – запрещает перевод процесса в состояние Ł Ł при выполнении операции открытия и любых последующих операциях над этим файлом;

O_APPEND – при открытии файла и перед выполнением каждой операции записи (если она, конечно, разрешена) указатель текущей позиции в файле устанавливается на конец файла;

O_TRUNC – если файл существует, уменьшить его размер до 0, с сохранением существующих атрибутов файла, кроме, быть может, времен последнего доступа к файлу и его последней модификации.

Кроме того, в некоторых версиях операционной системы UNIX могут применяться дополнительные значения флагов:

O_SYNC – любая операция записи в файл будет блокироваться (т. е. процесс будет переведен в состояние Ł Ł) до тех пор, пока записанная информация не будет физически помещена на соответсвующий нижестоящий уровень hardware;

O_NOCTTY – если имя файла относится к терминальному устройству, оно не становится управляющим терминалом процесса, даже если до этого процесс не имел управляющего терминала.

Параметр mode устанавливает атрибуты прав доступа различных категорий пользователей к новому файлу при его создании. Он обязателен, если среди заданных флагов присутствует флаг O_CREAT, и может быть опущен в противном случае. Этот параметр задается как сумма следующих восьмеричных значений:

0400 – разрешено чтение для пользователя, создавшего файл;

0200 – разрешена запись для пользователя, создавшего файл;

0100 – разрешено исполнение для пользователя, создавшего файл;

0040 – разрешено чтение для группы пользователя, создавшего файл;

0020 – разрешена запись для группы пользователя, создавшего файл;

0010 – разрешено исполнение для группы пользователя, создавшего файл;

0004 – разрешено чтение для всех остальных пользователей;

0002 – разрешена запись для всех остальных пользователей;

0001 – разрешено исполнение для всех остальных пользователей.

При создании файла реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode **и маски создания файлов текущего процесса** umask**, а именно — они равны** mode & ~umask**.**

При открытии файлов типа FIFO системный вызов имеет некоторые особенности поведения по сравнению с открытием файлов других типов. Если FIFO открывается только для чтения, и не задан флаг O_NDELAY**, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на запись. Если флаг** O_NDELAY **задан, то возвращается значение файлового дескриптора, ассоциированного с FIFO. Если FIFO открывается только для записи, и не задан флаг** O_NDELAY**, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на чтение. Если флаг** O_NDELAY **задан, то констатируется возникновение ошибки и возвращается значение** -1**.**

Возвращаемое значение

Системный вызов возвращает значение файлового дескриптора для открытого файла при нормальном завершении и значение -1 **при возникновении ошибки.**

Системный вызов open() использует набор флагов для того, чтобы специфицировать операции, которые предполагается применять к файлу в дальнейшем или которые должны быть выполнены непосредственно в момент открытия файла. Из всего возможного набора флагов на текущем уровне знаний нас будут интересовать только флаги O_RDONLY, O_WRONLY, O_RDWR, O_CREAT и O_EXCL. Первые три флага являются взаимоисключающими: хотя бы один из них должен быть применен и наличие одного из них не допускает наличия двух других. Эти флаги описывают набор операций, которые, при успешном открытии файла, будут разрешены над файлом в дальнейшем: только чтение, только запись, чтение и запись. Как вам известно из материалов семинаров 1–2, у каждого файла существуют атрибуты прав доступа для различных категорий пользователей. Если файл с заданным именем существует на диске, и права доступа к нему для пользователя, от имени которого работает текущий процесс, не противоречат запрошенному набору операций, то операционная система сканирует таблицу открытых файлов от ее начала к концу в поисках первого свободного элемента, заполняет его и возвращает индекс этого элемента в качестве файлового дескриптора открытого файла. Если файла на диске нет, не хватает прав или отсутствует свободное место в таблице открытых файлов, то констатируется возникновение ошибки.

В случае, когда мы **допускаем**, что файл на диске может отсутствовать, и хотим, чтобы он был создан, флаг для набора операций должен использоваться в комбинации с флагом O_CREAT. Если файл существует, то все происходит по рассмотренному выше сценарию. Если файла нет, сначала выполняется создание файла с набором прав, указанным в параметрах си-

стемного вызова. Проверка соответствия набора операций объявленным правам доступа может и не производиться (как, например, в Linux).

В случае, когда мы **требуем**, чтобы файл на диске отсутствовал и был создан в момент открытия, флаг для набора операций должен использоваться в комбинации с флагами O_CREAT и O_EXCL.

Подробнее об операции открытия файла и ее месте среди набора всех файловых операций будет рассказываться на лекции 7 «Файловая система с точки зрения пользователя». Работу системного вызова open() с флагами O_APPEND и O_TRUNC мы разберем на семинарах 11–12, посвященных организации файловых систем в UNIX.

Системные вызовы read(), write(), close()

Для совершения потоковых операций чтения информации из файла и ее записи в файл применяются системные вызовы read() и write().

Системные вызовы read и write

Прототипы системных вызовов

```
#include <sys/types.h>
#include <unistd.h>
size_t read(int fd, void *addr, size_t nbytes);
size_t write(int fd, void *addr, size_t nbytes);
```

Описание системных вызовов

Системные вызовы read и write предназначены для осуществления потоковых операций ввода (чтения) и вывода (записи) информации над каналами связи, описываемыми файловыми дескрипторами, т.е. для файлов, pipe, FIFO и socket.

Параметр fd является файловым дескриптором созданного ранее потокового канала связи, через который будет отсылаться или получаться информация, т. е. значением, которое вернул один из системных вызовов open(), pipe() или socket().

Параметр addr представляет собой адрес области памяти, начиная с которого будет браться информация для передачи или размещаться принятая информация.

Параметр nbytes для системного вызова write определяет количество байт, которое должно быть передано, начиная с адреса памяти addr. Параметр nbytes для системного вызова read определяет количество байт, которое мы хотим получить из канала связи и разместить в памяти, начиная с адреса addr.

Возвращаемые значения

В случае успешного завершения системный вызов возвращает количество реально посланных или принятых байт. Заметим, что это значение (большее или равное 0) может не совпадать с

заданным значением параметра nbytes, а быть меньше, чем оно, в силу отсутствия места на диске или в линии связи при передаче данных или отсутствия информации при ее приеме. При возникновении какой-либо ошибки возвращается отрицательное значение.

Особенности поведения при работе с файлами

При работе с файлами информация записывается в файл или читается из файла, начиная с места, определяемого указателем текущей позиции в файле. Значение указателя увеличивается на количество реально прочитанных или записанных байт. При чтении информации из файла она не пропадает из него. Если системный вызов read возвращает значение 0, то это означает, что файл прочитан до конца.

Мы сейчас не акцентируем внимание на понятии указателя текущей позиции в файле и взаимном влиянии значения этого указателя и поведения системных вызовов. Этот вопрос будет обсуждаться в дальнейшем на семинарах 11–12.

После завершения потоковых операций процесс должен выполнить операцию закрытия потока ввода-вывода, во время которой произойдет окончательный сброс буферов на линии связи, освободятся выделенные ресурсы операционной системы, и элемент таблицы открытых файлов, соответствующий файловому дескриптору, будет отмечен как свободный. За эти действия отвечает системный вызов close(). Надо отметить, что при завершении работы процесса (см. семинар 3–4) с помощью явного или неявного вызова функции exit() происходит автоматическое закрытие всех открытых потоков ввода-вывода.

Системный вызов close

Прототип системного вызова

```
#include <unistd.h>
int close(int fd);
```

Описание системного вызова

Системный вызов close предназначен для корректного завершения работы с файлами и другими объектами ввода-вывода, которые описываются в операционной системе через файловые дескрипторы: pipe, FIFO, socket.

Параметр fd является дескриптором соответствующего объекта, т. е. значением, которое вернул один из системных вызовов open(), pipe() или socket().

Возвращаемые значения

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Прогон программы для записи информации в файл

Для иллюстрации сказанного давайте рассмотрим следующую программу:

```
/* Программа 05-1.c, иллюстрирующая использование
системных вызовов open(), write() и close() для
записи информации в файл*/
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
int main(){
  int fd;
  size_t size;
  char string[] = "Hello, world!";
  /* Обнуляем маску создания файлов текущего процесса
  для того, чтобы права доступа у создаваемого файла
  точно соответствовали параметру вызова open() */
  (void)umask(0);
  /* Попытаемся открыть файл с именем myfile в текущей
  директории только для операций вывода. Если файла
  не существует, попробуем его создать с правами дос-
  тупа 0666, т. е. read-write для всех категорий
  пользователей */
  if((fd = open("myfile", O_WRONLY | O_CREAT,
    0666)) < 0){
    /* Если файл открыть не удалось, печатаем об
    этом сообщение и прекращаем работу */
    printf("Can\'t open file\n");
    exit(-1);
  }
  /* Пробуем записать в файл 14 байт из нашего
  массива, т.е. всю строку "Hello, world!" вместе
  с признаком конца строки */
  size = write(fd, string, 14);
  if(size != 14){
    /* Если записалось меньшее количество байт,
    сообщаем об ошибке */
    printf("Can\'t write all string\n");
    exit(-1);
  }
```

```
    /* Закрываем файл */
    if(close(fd) < 0){
      printf("Can\'t close file\n");
    }
    return 0;
}
```

Наберите, откомпилируйте эту программу и запустите ее на исполнение. Обратите внимание на использование системного вызова `umask()` с параметром `0` для того, чтобы права доступа к созданному файлу точно соответствовали указанным в системном вызове `open()`.

Написание, компиляция и запуск программы для чтения информации из файла

Измените программу из предыдущего раздела так, чтобы она читала записанную ранее в файл информацию и печатала ее на экране. Все лишние операторы желательно удалить.

Понятие о pipe. Системный вызов pipe()

Наиболее простым способом для передачи информации с помощью потоковой модели между различными процессами или даже внутри одного процесса в операционной системе UNIX является pipe (канал, труба, конвейер).

Важное отличие pip'а от файла заключается в том, что прочитанная информация немедленно удаляется из него и не может быть прочитана повторно.

Pipe можно представить себе в виде трубы ограниченной емкости, расположенной внутри адресного пространства операционной системы, доступ к входному и выходному отверстию которой осуществляется с помощью системных вызовов. В действительности pipe представляет собой область памяти, недоступную пользовательским процессам напрямую, зачастую организованную в виде кольцевого буфера (хотя существуют и другие виды организации). По буферу при операциях чтения и записи перемещаются два указателя, соответствующие входному и выходному потокам. При этом выходной указатель никогда не может перегнать входной и наоборот. Для создания нового экземпляра такого кольцевого буфера внутри операционной системы используется системный вызов `pipe()`.

Системный вызов pipe

Прототип системного вызова

```
#include <unistd.h>
int pipe(int *fd);
```

Описание системного вызова

Системный вызов `pipe` предназначен для создания pip'а внутри операционной системы.

Параметр `fd` является указателем на массив из двух целых переменных. При нормальном завершении вызова в первый элемент массива – `fd[0]` – будет занесен файловый дескриптор, соответствующий выходному потоку данных pip'а и позволяющий выполнять только операцию чтения, а во второй элемент массива – `fd[1]` – будет занесен файловый дескриптор, соответствующий входному потоку данных и позволяющий выполнять только операцию записи.

Возвращаемые значения

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибок.

В процессе работы системный вызов организует выделение области памяти под буфер и указатели и заносит информацию, соответствующую входному и выходному потокам данных, в два элемента таблицы открытых файлов, связывая тем самым с каждым pip'ом два файловых дескриптора. Для одного из них разрешена только операция чтения из pip'а, а для другого – только операция записи в pipe. Для выполнения этих операций мы можем использовать те же самые системные вызовы `read()` и `write()`, что и при работе с файлами. Естественно, по окончании использования входного или/и выходного потока данных, нужно закрыть соответствующий поток с помощью системного вызова `close()` для освобождения системных ресурсов. Необходимо отметить, что, когда все процессы, использующие pipe, закрывают все ассоциированные с ним файловые дескрипторы, операционная система ликвидирует pipe. Таким образом, время существования pip'а в системе не может превышать время жизни процессов, работающих с ним.

Прогон программы для pipe в одном процессе

Достаточно яркой иллюстрацией действий по созданию pip'a, записи в него данных, чтению из него и освобождению выделенных ресурсов может служить программа, организующая работу с pip'ом в рамках одного процесса, приведенная ниже.

```
/* Программа 05-2.c, иллюстрирующая работу с pip'ом
в рамках одного процесса */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
  int fd[2];
  size_t size;
  char string[] = "Hello, world!";
  char resstring[14];
  /* Попытаемся создать pipe */
  if(pipe(fd) < 0){
    /* Если создать pipe не удалось, печатаем об
    этом сообщение и прекращаем работу */
    printf("Can\'t create pipe\n");
    exit(-1);
  }
  /* Пробуем записать в pipe 14 байт из нашего
  массива, т.е. всю строку "Hello, world!" вместе
  с признаком конца строки */
  size = write(fd[1], string, 14);
  if(size != 14){
    /* Если записалось меньшее количество байт,
    сообщаем об ошибке */
    printf("Can\'t write all string\n");
    exit(-1);
  }
  /* Пробуем прочитать из pip'а 14 байт в другой
  массив, т.е. всю записанную строку */
  size = read(fd[0], resstring, 14);
  if(size < 0){
    /* Если прочитать не смогли, сообщаем об ошибке */
    printf("Can\'t read string\n");
    exit(-1);
  }
  /* Печатаем прочитанную строку */
  printf("%s\n",resstring);
  /* Закрываем входной поток*/
  if(close(fd[0]) < 0){
    printf("Can\'t close input stream\n");
  }
```

```
    /* Закрываем выходной поток*/
    if(close(fd[1]) < 0){
      printf("Can\'t close output stream\n");
    }
    return 0;
}
```

Наберите программу, откомпилируйте ее и запустите на исполнение.

Организация связи через pipe между процессом-родителем и процессом-потомком. Наследование файловых дескрипторов при вызовах fork() и exec()

Понятно, что если бы все достоинство pip'ов сводилось к замене функции копирования из памяти в память внутри одного процесса на пересылку информации через операционную систему, то овчинка не стоила бы выделки. Однако таблица открытых файлов наследуется процессом-ребенком при порождении нового процесса системным вызовом `fork()` и входит в состав неизменяемой части системного контекста процесса при системном вызове `exec()` (за исключением тех потоков данных, для файловых дескрипторов которых был специальными средствами выставлен признак, побуждающий операционную систему закрыть их при выполнении `exec()`, однако их рассмотрение выходит за рамки нашего курса). Это обстоятельство позволяет организовать передачу информации через pipe между родственными процессами, имеющими общего прародителя, создавшего pipe.

Прогон программы для организации однонаправленной связи между родственными процессами через pipe

Давайте рассмотрим программу, осуществляющую однонаправленную связь между процессом-родителем и процессом-ребенком:

```
/* Программа 05-3.с, осуществляющая однонаправленную
связь через pipe между процессом-родителем и
процессом-ребенком */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
```

```
int main(){
  int fd[2], result;
  size_t size;
  char resstring[14];
  /* Попытаемся создать pipe */
  if(pipe(fd) < 0){
    /* Если создать pipe не удалось, печатаем об
    этом сообщение и прекращаем работу */
    printf("Can\'t create pipe\n");
    exit(-1);
  }
  /* Порождаем новый процесс */
  result = fork();
  if(result){
    /* Если создать процесс не удалось, сообщаем
    об этом и завершаем работу */
    printf("Can\'t fork child\n");
    exit(-1);
  } else if (result > 0) {
    /* Мы находимся в родительском процессе, который
    будет передавать информацию процессу-ребенку. В
    этом процессе выходной поток данных нам
    не понадобится, поэтому закрываем его.*/
    close(fd[0]);
    /* Пробуем записать в pipe 14 байт, т.е. всю
    строку "Hello, world!" вместе с признаком конца
    строки */
    size = write(fd[1], "Hello, world!", 14);
    if(size != 14){
      /* Если записалось меньшее количество байт,
      сообщаем об ошибке и завершаем работу */
      printf("Can\'t write all string\n");
      exit(-1);
    }
    /* Закрываем входной поток данных, на этом
    родитель прекращает работу */
    close(fd[1]);
    printf("Parent exit\n");
  } else {
    /* Мы находимся в порожденном процессе, который
    будет получать информацию от процесса-родителя.
    Он унаследовал от родителя таблицу открытых файлов
```

```
        и, зная файловые дескрипторы, соответствующие
        pip'у, может его использовать. В этом процессе
        входной поток данных нам не понадобится, поэтому
        закрываем его.*/
        close(fd[1]);
        /* Пробуем прочитать из pip'а 14 байт в массив,
        т.е. всю записанную строку */
        size = read(fd[0], resstring, 14);
        if(size < 0){

           /* Если прочитать не смогли, сообщаем об
           ошибке и завершаем работу */

           printf("Can\'t read string\n");
           exit(-1);
        }
        /* Печатаем прочитанную строку */
        printf("%s\n",resstring);
        /* Закрываем входной поток и завершаем работу */
        close(fd[0]);
    }
    return 0;
}
```

Наберите программу, откомпилируйте ее и запустите на исполнение.

Задача повышенной сложности: модифицируйте этот пример для связи между собой двух родственных процессов, исполняющих разные программы.

Написание, компиляция и запуск программы для организации двунаправленной связи между родственными процессами через pipe

Pipe служит для организации однонаправленной или симплексной связи. Если бы в предыдущем примере мы попытались организовать через pipe двустороннюю связь, когда процесс-родитель пишет информацию в pipe, предполагая, что ее получит процесс-ребенок, а затем читает информацию из pipe, предполагая, что ее записал порожденный процесс, то могла бы возникнуть ситуация, в которой процесс-родитель прочитал бы собственную информацию, а процесс-ребенок не получил бы ничего. Для использования одного pipe в двух направлениях необходимы специальные средства синхронизации

процессов, о которых речь идет в лекциях «Алгоритмы синхронизации» (лекция 5) и «Механизмы синхронизации» (лекция 6). Более простой способ организации двунаправленной связи между родственными процессами заключается в использовании двух pipe. Модифицируйте программу из предыдущего примера (раздел «Прогон программы для организации однонаправленной связи между родственными процессами через pipe») для организации такой двусторонней связи, откомпилируйте ее и запустите на исполнение.

Необходимо отметить, что в некоторых UNIX-подобных системах (например, в Solaris 2) реализованы полностью дуплексные pip'ы [Стивенс, 2002]. В таких системах для обоих файловых дескрипторов, ассоциированных с pipe, разрешены и операция чтения, и операция записи. Однако такое поведение не характерно для pip'ов и не является переносимым.

Особенности поведения вызовов read() и write() для pip'а

Системные вызовы `read()` и `write()` имеют определенные особенности поведения при работе с pipe, связанные с его ограниченным размером, задержками в передаче данных и возможностью блокирования обменивающихся информацией процессов. Организация запрета блокирования этих вызовов для pipe выходит за рамки нашего курса.

Будьте внимательны при написании программ, обменивающихся большими объемами информации через pipe. Помните, что за один раз из pipe может прочитаться меньше информации, чем вы запрашивали, и за один раз в pipe может записаться меньше информации, чем вам хотелось бы. Проверяйте значения, возвращаемые вызовами!

Одна из особенностей поведения блокирующегося системного вызова `read()` связана с попыткой чтения из пустого pipe. Если есть процессы, у которых этот pipe открыт для записи, то системный вызов блокируется и ждет появления информации. Если таких процессов нет, он вернет значение 0 без блокировки процесса. Эта особенность приводит к **необходимости закрытия файлового дескриптора, ассоциированного с входным концом pipe, в процессе, который будет использовать pipe для чтения** (`close(fd[1])`) в процессе-ребенке в программе из раздела «Прогон программы для организации однонаправленной связи между родственными процессами через pipe»). Аналогичной особенностью поведения при отсутствии процессов, у которых pipe открыт для чтения, обладает и системный вызов `write()`, с чем связана **необходимость закрытия файлового дескриптора, ассоциированного с выходным концом pipe, в процессе, который будет использовать pipe для записи** (`close(fd[0])`) в процессе-родителе в той же программе).

Системные вызовы read и write (продолжение)

Особенности поведения при работе с pipe, FIFO и socket

Системный вызов read

Ситуация	Поведение
Попытка прочитать меньше байт, чем есть в наличии в канале связи.	Читает требуемое количество байт и возвращает значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи.
В канале связи находится меньше байт, чем затребовано, но не нулевое количество.	Читает все, что есть в канале связи, и возвращает значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи.
Попытка читать из канала связи, в котором нет информации. Блокировка вызова разрешена.	Вызов блокируется до тех пор, пока не появится информация в канале связи и пока существует процесс, который может передать в него информацию. Если информация появилась, то процесс разблокируется, и поведение вызова определяется двумя предыдущими строками таблицы. Если в канал некому передать данные (нет ни одного процесса, у которого этот канал связи открыт для записи), то вызов возвращает значение `0`. Если канал связи полностью закрывается для записи во время блокировки читающего процесса, то процесс разблокируется, и системный вызов возвращает значение `0`.
Попытка читать из канала связи, в котором нет информации. Блокировка вызова не разрешена.	Если есть процессы, у которых канал связи открыт для записи, системный вызов возвращает значение `-1` и устанавливает переменную `errno` в значение `EAGAIN`. Если таких процессов нет, системный вызов возвращает значение `0`.

Системный вызов write

Ситуация	Поведение
Попытка записать в канал связи меньше байт, чем осталось до его заполнения.	Требуемое количество байт помещается в канал связи, возвращается записанное количество байт.
Попытка записать в канал связи больше байт, чем осталось до его заполнения. Блокировка вызова разрешена.	Вызов блокируется до тех пор, пока все данные не будут помещены в канал связи. Если размер буфера канала связи меньше, чем передаваемое количество информации, то вызов будет ждать, пока часть информации не будет считана из канала связи. Возвращается записанное количество байт.

Попытка записать в канал связи больше байт, чем осталось до его заполнения, но меньше, чем размер буфера канала связи. Блокировка вызова запрещена.	Системный вызов возвращает значение -1 и устанавливает переменную errno в значение EAGAIN.
В канале связи есть место. Попытка записать в канал связи больше байт, чем осталось до его заполнения, и больше, чем размер буфера канала связи. Блокировка вызова запрещена.	Записывается столько байт, сколько осталось до заполнения канала. Системный вызов возвращает количество записанных байт.
Попытка записи в канал связи, в котором нет места. Блокировка вызова не разрешена.	Системный вызов возвращает значение -1 и устанавливает переменную errno в значение EAGAIN.
Попытка записи в канал связи, из которого некому больше читать, или полное закрытие канала на чтение во время блокировки системного вызова.	Если вызов был заблокирован, то он разблокируется. Процесс получает сигнал SIGPIPE. Если этот сигнал обрабатывается пользователем, то системный вызов вернет значение -1 и установит переменную errno в значение EPIPE.

Необходимо отметить дополнительную особенность системного вызова write при работе с pip'ами и FIFO. Запись информации, размер которой не превышает размер буфера, должна осуществляться атомарно – одним подряд лежащим куском. Этим объясняется ряд блокировок и ошибок в предыдущем перечне.

Задача повышенной сложности: определите размер pipe для вашей операционной системы.

Понятие FIFO. Использование системного вызова mknod() для создания FIFO. Функция mkfifo()

Как мы выяснили, доступ к информации о расположении pip'a в операционной системе и его состоянии может быть осуществлен только через таблицу открытых файлов процесса, создавшего pipe, и через унаследованные от него таблицы открытых файлов процессов-потомков. Поэтому изложенный выше механизм обмена информацией через pipe справедлив лишь для родственных процессов, имеющих общего прародителя, инициировавшего системный вызов pipe(), или для таких процессов и самого прародителя и не может использоваться для потокового общения с другими процессами. В операционной системе UNIX существует возможность использования pip'a для взаимодействия других процессов, но

ее реализация достаточно сложна и лежит далеко за пределами наших занятий.

Для организации потокового взаимодействия любых процессов в операционной системе UNIX применяется средство связи, получившее название FIFO (от **F**irst **I**nput **F**irst **O**utput) или именованный pipe. FIFO во всем подобен pip'у, за одним исключением: данные о расположении FIFO в адресном пространстве ядра и его состоянии процессы могут получать не через родственные связи, а через файловую систему. Для этого при создании именованного pip'а на диске заводится файл специального типа, обращаясь к которому процессы могут получить интересующую их информацию. Для создания FIFO используется системный вызов `mknod()` или существующая в некоторых версиях UNIX функция `mkfifo()`. **Следует отметить, что при их работе не происходит действительного выделения области адресного пространства операционной системы под именованный pipe, а только заводится файл-метка, существование которой позволяет осуществить реальную организацию FIFO в памяти при его открытии с помощью уже известного нам ситемного вызова `open()`.** После открытия именованный pipe ведет себя точно так же, как и неименованный. Для дальнейшей работы с ним применяются системные вызовы `read()`, `write()` и `close()`. Время существования FIFO в адресном пространстве ядра операционной системы, как и в случае с pip'ом, не может превышать время жизни последнего из использовавших его процессов. Когда все процессы, работающие с FIFO, закрывают все файловые дескрипторы, ассоциированные с ним, система освобождает ресурсы, выделенные под FIFO. Вся непрочитанная информация теряется. В то же время файл-метка остается на диске и может использоваться для новой реальной организации FIFO в дальнейшем.

Использование системного вызова mknod для создания FIFO

Прототип системного вызова

```
#include <sys/stat.h>
#include <unistd.h>

int mknod(char *path, int mode, int dev);
```

Описание системного вызова

Нашей целью является не полное описание системного вызова `mknod`, а только описание его использования для создания FIFO. Поэтому мы будем рассматривать не все возможные варианты задания параметров, а только те из них, которые соответствуют этой специфической деятельности.

Параметр dev является несущественным в нашей ситуации, и мы будем всегда задавать его равным 0.

Параметр path является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом существовать не должно.

Параметр mode устанавливает атрибуты прав доступа различных категорий пользователей к FIFO. Этот параметр задается как результат побитовой операции «или» значения S_IFIFO, указывающего, что системный вызов должен создать FIFO, и некоторой суммы следующих восьмеричных значений:

0400 – разрешено чтение для пользователя, создавшего FIFO;
0200 – разрешена запись для пользователя, создавшего FIFO;
0040 – разрешено чтение для группы пользователя, создавшего FIFO;
0020 – разрешена запись для группы пользователя, создавшего FIFO;
0004 – разрешено чтение для всех остальных пользователей;
0002 – разрешена запись для всех остальных пользователей.

При создании FIFO реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса umask, а именно – они равны (0777 & mode) & ~umask.

Возвращаемые значения

При успешном создании FIFO системный вызов возвращает значение 0, при неуспешном – отрицательное значение.

Функция mkfifo

Прототип функции

```
#include <sys/stat.h>
#include <unistd.h>

int mkfifo(char *path, int mode);
```

Описание функции

Функция mkfifo предназначена для создания FIFO в операционной системе.

Параметр path является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом функции не должно существовать.

Параметр mode устанавливает атрибуты прав доступа различных категорий пользователей к FIFO. Этот параметр задается как некоторая сумма следующих восьмеричных значений:

0400 – разрешено чтение для пользователя, создавшего FIFO;
0200 – разрешена запись для пользователя, создавшего FIFO;

0040 – разрешено чтение для группы пользователя, создавшего FIFO;
0020 – разрешена запись для группы пользователя, создавшего FIFO;
0004 – разрешено чтение для всех остальных пользователей;
0002 – разрешена запись для всех остальных пользователей.

При создании FIFO реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса umask, а именно – они равны (0777 & mode) & ~umask.

Возвращаемые значения

При успешном создании FIFO функция возвращает значение 0, при неуспешном – отрицательное значение.

Важно понимать, что файл типа FIFO не служит для размещения на диске информации, которая записывается в именованный pipe. Эта информация располагается внутри адресного пространства операционной системы, а файл является только меткой, создающей предпосылки для ее размещения.

Не пытайтесь просмотреть содержимое этого файла с помощью Midnight Commander (mc)!!! Это приведет к его глубокому зависанию!

Особенности поведения вызова open() при открытии FIFO

Системные вызовы read() и write() при работе с FIFO имеют те же особенности поведения, что и при работе с pip'ом. Системный вызов open() при открытии FIFO также ведет себя несколько иначе, чем при открытии других типов файлов, что связано с возможностью блокирования выполняющих его процессов. Если FIFO открывается только для чтения, и флаг O_NDELAY не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на запись. Если флаг O_NDELAY задан, то возвращается значение файлового дескриптора, ассоциированного с FIFO. Если FIFO открывается только для записи, и флаг O_NDELAY не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на чтение. Если флаг O_NDELAY задан, то констатируется возникновение ошибки и возвращается значение -1. Задание флага O_NDELAY в параметрах системного вызова open() приводит и к тому, что процессу, открывшему FIFO, запрещается блокировка при выполнении последующих операций чтения из этого потока данных и записи в него.

Прогон программы с FIFO в родственных процессах

Для иллюстрации взаимодействия процессов через FIFO рассмотрим такую программу:

```
/* Программа 05-4.c, осуществляющая однонаправленную
связь через FIFO между процессом-родителем и
процессом-ребенком */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main(){
  int fd, result; size_t size; char resstring[14];
  char name[]="aaa.fifo";
  /* Обнуляем маску создания файлов текущего процесса
  для того, чтобы права доступа у создаваемого FIFO
  точно соответствовали параметру вызова mknod() */
  (void)umask(0);
  /* Попытаемся создать FIFO с именем aaa.fifo в
  текущей директории */
  if(mknod(name, S_IFIFO | 0666, 0) < 0){
    /* Если создать FIFO не удалось, печатаем об
    этом сообщение и прекращаем работу */
    printf("Can\'t create FIFO\n");
    exit(-1);
  }
  /* Порождаем новый процесс */
  if((result = fork()) < 0){
    /* Если создать процесс не удалось, сообщаем
    об этом и завершаем работу */
    printf("Can\'t fork child\n");
    exit(-1);
  } else if (result > 0) {
    /* Мы находимся в родительском процессе, который
    будет передавать информацию процессу-ребенку. В
    этом процессе открываем FIFO на запись.*/
    if((fd = open(name, O_WRONLY)) < 0){
      /* Если открыть FIFO не удалось, печатаем
      об этом сообщение и прекращаем работу */
      printf("Can\'t open FIFO for writing\n");
```

```
        exit(-1);
    }
    /* Пробуем записать в FIFO 14 байт, т.е. всю
    строку "Hello, world!" вместе с признаком конца
    строки */
    size = write(fd, "Hello, world!", 14);
    if(size != 14){
    /* Если записалось меньшее количество байт, то
    сообщаем об ошибке и завершаем работу */
        printf("Can\'t write all string to FIFO\n");
        exit(-1);
    }
    /* Закрываем входной поток данных и на этом
    родитель прекращает работу */
    close(fd);
    printf("Parent exit\n");
  } else {
    /* Мы находимся в порожденном процессе, который
    будет получать информацию от процесса-родителя.
    Открываем FIFO на чтение.*/
    if((fd = open(name, O_RDONLY)) < 0){
        /* Если открыть FIFO не удалось, печатаем об
        этом сообщение и прекращаем работу */
        printf("Can\'t open FIFO for reading\n");
        exit(-1);
    }
    /* Пробуем прочитать из FIFO 14 байт в массив,
    т.е. всю записанную строку */
    size = read(fd, resstring, 14);
    if(size < 0){
        /* Если прочитать не смогли, сообщаем об ошибке
        и завершаем работу */
        printf("Can\'t read string\n");
        exit(-1);
    }
    /* Печатаем прочитанную строку */
    printf("%s\n",resstring);
    /* Закрываем входной поток и завершаем работу */
    close(fd);
  }
  return 0;
}
```

Наберите программу, откомпилируйте ее и запустите на исполнение. В этой программе информацией между собой обмениваются процесс-родитель и процесс-ребенок. Обратим внимание, что повторный запуск этой программы приведет к ошибке при попытке создания FIFO, так как файл с заданным именем уже существует. Здесь нужно либо удалять его перед каждым прогоном программы с диска вручную, либо после первого запуска модифицировать исходный текст, исключив из него все, связанное с системным вызовом `mknod()`. С системным вызовом, предназначенным для удаления файла при работе процесса, мы познакомимся позже (на семинарах 11—12) при изучении файловых систем.

Написание, компиляция и запуск программы с FIFO в неродственных процессах

Для закрепления полученных знаний напишите на базе предыдущего примера две программы, одна из которых пишет информацию в FIFO, а вторая – читает из него, так чтобы между ними не было ярко выраженных родственных связей (т. е. чтобы ни одна из них не была потомком другой).

Неработающий пример для связи процессов на различных компьютерах

Если у вас есть возможность, найдите два компьютера, имеющие разделяемую файловую систему (например, смонтированную с помощью NFS), и запустите на них программы из предыдущего раздела так, чтобы каждая программа работала на своем компьютере, а FIFO создавалось на разделяемой файловой системе. Хотя оба процесса видят один и тот же файл с типом FIFO, взаимодействия между ними не происходит, так как они функционируют в физически разных адресных пространствах и пытаются открыть FIFO внутри различных операционных систем.

Семинары 6–7. Средства System V IPC. Организация работы с разделяемой памятью в UNIX. Понятие нитей исполнения (thread)

Преимущества и недостатки потокового обмена данными. Понятие System V IPC. Пространство имен. Адресация в System V IPC. Функция ftok(). Дескрипторы System V IPC. Разделяемая память в UNIX. Системные вызовы shmget(), shmat(), shmdt(). Команды ipc и ipcrm. Использование системного вызова shmctl() для освобождения ресурса. Разделяемая память и системные вызовы fork(), exec() и функция exit(). Понятие о нити исполнения (thread) в UNIX. Идентификатор нити исполнения. Функция pthread_self(). Создание и завершение thread'а. Функции pthread_create(), pthread_exit(), pthread_join(). Необходимость синхронизации процессов и нитей исполнения, использующих общую память.

Ключевые слова: System V IPC, пространство имен средств связи, ключ System V IPC, дескриптор (идентификатор) System V IPC, разделяемая память (shared memory), функция ftok, системные вызовы shmget, shmat, shmdt, shmctl, флаги IPC_CREAT и IPC_EXCL, ключ IPC_PRIVATE, команды ipcrm и ipcs, нить исполнения (thread), идентификатор нити исполнения (TID), библиотека pthread, функции pthread_self, pthread_create, pthread_exit, pthread_join.

Преимущества и недостатки потокового обмена данными

На предыдущем семинаре мы познакомились с механизмами, обеспечивающими потоковую передачу данных между процессами в операционной системе UNIX, а именно с pip'ами и FIFO. Потоковые механизмы достаточно просты в реализации и удобны для использования, но имеют ряд существенных недостатков:

- Операции чтения и записи не анализируют содержимое передаваемых данных. Процесс, прочитавший 20 байт из потока, не может сказать, были ли они записаны одним процессом или несколькими, записывались ли они за один раз или было, например, выполнено 4 операции записи по 5 байт. Данные в потоке никак не интерпретируются системой. Если требуется какая-либо интерпретация данных, то передающий и принимающий процессы должны заранее согласовать свои действия и уметь осуществлять ее самостоятельно.

- Для передачи информации от одного процесса к другому требуется, как минимум, две операции копирования данных: первый раз – из адресного пространства передающего процесса в системный буфер, второй раз – из системного буфера в адресное пространство принимающего процесса.
- Процессы, обменивающиеся информацией, должны одновременно существовать в вычислительной системе. Нельзя записать информацию в поток с помощью одного процесса, завершить его, а затем, через некоторое время, запустить другой процесс и прочитать записанную информацию.

Понятие о System V IPC

Указанные выше недостатки потоков данных привели к разработке других механизмов передачи информации между процессами. Часть этих механизмов, впервые появившихся в UNIX System V и впоследствии перекочевавших оттуда практически во все современные версии операционной системы UNIX, получила общее название *System V IPC* (IPC – сокращение от InterProcess Communications). В группу System V IPC входят: очереди сообщений, разделяемая память и семафоры. Эти средства организации взаимодействия процессов связаны не только общностью происхождения, но и обладают схожим интерфейсом для выполнения подобных операций, например, для выделения и освобождения соответствующего ресурса в системе. Мы будем рассматривать их в порядке от менее семантически нагруженных с точки зрения операционной системы к более семантически нагруженным. Иными словами, чем позже мы начнем заниматься каким-либо механизмом из System V IPC, тем больше действий по интерпретации передаваемой информации придется выполнять операционной системе при использовании этого механизма. Часть этого семинара мы посвятим изучению разделяемой памяти. Семафоры будут рассматриваться на семинаре 8, а очереди сообщений – на семинаре 9.

Пространство имен. Адресация в System V IPC. Функция ftok()

Все средства связи из System V IPC, как и уже рассмотренные нами pipe и FIFO, являются средствами связи с непрямой адресацией. Как мы установили на предыдущем семинаре, для организации взаимодействия неродственных процессов с помощью средства связи с непрямой адресацией необходимо, чтобы это средство связи имело имя. Отсутствие имен у pip'ов позволяет процессам получать информацию о расположении pip'а в системе и его состоянии только через родственные связи. Наличие

ассоциированного имени у FIFO – имени специализированного файла в файловой системе – позволяет неродственным процессам получать эту информацию через интерфейс файловой системы.

Множество всех возможных имен для объектов какого-либо вида принято называть пространством имен соответствующего вида объектов. Для FIFO пространством имен является множество всех допустимых имен файлов в файловой системе. Для всех объектов из System V IPC таким пространством имен является множество значений некоторого целочисленного типа данных – `key_t` – ключа. Причем программисту не позволено напрямую присваивать значение ключа, это значение задается опосредованно: через комбинацию имени какого-либо файла, уже существующего в файловой системе, и небольшого целого числа – например, номера экземпляра средства связи.

Такой хитрый способ получения значения ключа связан с двумя соображениями:

- Если разрешить программистам самим присваивать значение ключа для идентификации средств связи, то не исключено, что два программиста случайно воспользуются одним и тем же значением, не подозревая об этом. Тогда их процессы будут несанкционированно взаимодействовать через одно и то же средство коммуникации, что может привести к нестандартному поведению этих процессов. Поэтому основным компонентом значения ключа является преобразованное в числовое значение полное имя некоторого файла, доступ к которому на чтение разрешен процессу. Каждый программист имеет возможность использовать для этой цели свой специфический файл, например исполняемый файл, связанный с одним из взаимодействующих процессов. Следует отметить, что преобразование из текстового имени файла в число основывается на расположении указанного файла на жестком диске или ином физическом носителе. Поэтому для образования ключа следует применять файлы, не меняющие своего положения в течение времени организации взаимодействия процессов.
- Второй компонент значения ключа используется для того, чтобы позволить программисту связать с одним и тем же именем файла более одного экземпляра каждого средства связи. В качестве такого компонента можно задавать порядковый номер соответствующего экземпляра.

Получение значения ключа из двух компонентов осуществляется функцией `ftok()`.

Функция для генерации ключа System V IPC

Прототип функции

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *pathname, char proj);
```

Описание функции

Функция `ftok` служит для преобразования имени существующего файла и небольшого целого числа, например, порядкового номера экземпляра средств связи, в ключ System V IPC.

Параметр `pathname` должен являться указателем на имя существующего файла, доступного для процесса, вызывающего функцию.

Параметр `proj` – это небольшое целое число, характеризующее экземпляр средства связи.

В случае невозможности генерации ключа функция возвращает отрицательное значение, в противном случае она возвращает значение сгенерированного ключа. Тип данных `key_t` обычно представляет собой 32-битовое целое.

Еще раз подчеркнем три важных момента, связанных с использованием имени файла для получения ключа. Во-первых, необходимо указывать имя файла, который **уже существует** в файловой системе и для которого процесс **имеет право доступа на чтение** (не путайте с заданием имени файла при создании FIFO, где указывалось имя **для вновь создаваемого** специального файла). Во-вторых, указанный файл должен **сохранять свое положение на диске** до тех пор, пока все процессы, участвующие во взаимодействии, не получат ключ System V IPC. В-третьих, задание имени файла, как одного из компонентов для получения ключа, ни в коем случае не означает, что информация, передаваемая с помощью ассоциированного средства связи, будет располагаться в этом файле. Информация будет храниться **внутри адресного пространства операционной системы**, а заданное имя файла лишь позволяет различным процессам сгенерировать идентичные ключи.

Дескрипторы System V IPC

Мы говорили (см. семинар 5, раздел «Файловый дескриптор»), что информацию о потоках ввода-вывода, с которыми имеет дело текущий процесс, в частности о pip'ах и FIFO, операционная система хранит в таблице открытых файлов процесса. Системные вызовы, осуществляющие операции над потоком, используют в качестве параметра индекс элемента таблицы открытых файлов, соответствующий потоку, – файловый деск-

риптор. Использование файловых дескрипторов для идентификации потоков внутри процесса позволяет применять к ним уже существующий интерфейс для работы с файлами, но в то же время приводит к автоматическому закрытию потоков при завершении процесса. Этим, в частности, объясняется один из перечисленных выше недостатков потоковой передачи информации.

При реализации компонентов System V IPC была принята другая концепция. Ядро операционной системы хранит информацию обо всех средствах System V IPC, используемых в системе, вне контекста пользовательских процессов. При создании нового средства связи или получении доступа к уже существующему процесс получает неотрицательное целое число – *дескриптор* (*идентификатор*) *этого средства связи*, который однозначно идентифицирует его во всей вычислительной системе. Этот дескриптор должен передаваться в качестве параметра всем системным вызовам, осуществляющим дальнейшие операции над соответствующим средством System V IPC.

Подобная концепция позволяет устранить один из самых существенных недостатков, присущих потоковым средствам связи – требование одновременного существования взаимодействующих процессов, но в то же время требует повышенной осторожности для того, чтобы процесс, получающий информацию, не принял взамен новых старые данные, случайно оставленные в механизме коммуникации.

Разделяемая память в UNIX. Системные вызовы shmget(), shmat(), shmdt()

С точки зрения операционной системы, наименее семантически нагруженным средством System V IPC является разделяемая память (shared memory). Мы уже упоминали об этой категории средств связи на лекции. Для текущего семинара нам достаточно знать, что операционная система может позволить нескольким процессам совместно использовать некоторую область адресного пространства. Внутренние механизмы, позволяющие реализовать такое использование, будут подробно рассмотрены на лекции, посвященной сегментной, страничной и сегментно-страничной организации памяти.

Все средства связи System V IPC требуют предварительных инициализирующих действий (создания) для организации взаимодействия процессов.

Для создания области разделяемой памяти с определенным ключом или доступа по ключу к уже существующей области применяется системный вызов `shmget()`. Существует два варианта его использования для создания новой области разделяемой памяти:

- Стандартный способ. В качестве значения ключа системному вызову поставляется значение, сформированное функцией `ftok()` для некоторого имени файла и номера экземпляра области разделяемой памяти. В качестве флагов поставляется комбинация прав доступа к создаваемому сегменту и флага `IPC_CREAT`. Если сегмент для данного ключа еще не существует, то система будет пытаться создать его с указанными правами доступа. Если же вдруг он уже существовал, то мы просто получим его дескриптор. Возможно добавление к этой комбинации флагов флага `IPC_EXCL`. Этот флаг гарантирует нормальное завершение системного вызова только в том случае, если сегмент действительно был создан (т. е. ранее он не существовал), если же сегмент существовал, то системный вызов завершится с ошибкой, и значение системной переменной `errno`, описанной в файле `errno.h`, будет установлено в `EEXIST`.
- Нестандартный способ. В качестве значения ключа указывается специальное значение `IPC_PRIVATE`. Использование значения `IPC_PRIVATE` **всегда** приводит к попытке создания нового сегмента разделяемой памяти с заданными правами доступа и с ключом, который не совпадает со значением ключа ни одного из уже существующих сегментов и который не может быть получен с помощью функции `ftok()` ни при одной комбинации ее параметров. Наличие флагов `IPC_CREAT` и `IPC_EXCL` в этом случае игнорируется.

Системный вызов shmget()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg);
```

Описание системного вызова

Системный вызов `shmget` предназначен для выполнения операции доступа к сегменту разделяемой памяти и, в случае его успешного завершения, возвращает дескриптор System V IPC для этого сегмента (целое неотрицательное число, однозначно характеризующее сегмент внутри вычислительной системы и использующееся в дальнейшем для других операций с ним).

Параметр `key` является ключом System V IPC для сегмента, т. е. фактически его именем из пространства имен System V IPC. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции `ftok()`, или специальное значение `IPC_PRIVATE`. Использование значения `IPC_PRIVATE` **всегда** приводит к попытке создания

нового сегмента разделяемой памяти с ключом, который не совпадает со значением ключа ни одного из уже существующих сегментов и который не может быть получен с помощью функции ftok() ни при одной комбинации ее параметров.

Параметр size определяет размер создаваемого или уже существующего сегмента в байтах. Если сегмент с указанным ключом уже существует, но его размер не совпадает с указанным в параметре size, констатируется возникновение ошибки.

Параметр shmflg – флаги – играет роль только при создании нового сегмента разделяемой памяти и определяет права различных пользователей при доступе к сегменту, а также необходимость создания нового сегмента и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или – « | ») следующих предопределенных значений и восьмеричных прав доступа:

IPC_CREAT – если сегмента для указанного ключа не существует, он должен быть создан;
IPC_EXCL – применяется совместно с флагом IPC_CREAT. При совместном их использовании и существовании сегмента с указанным ключом, доступ к сегменту не производится и констатируется ошибочная ситуация, при этом переменная errno, описанная в файле <errno.h>, примет значение EEXIST;
0400 – разрешено чтение для пользователя, создавшего сегмент;
0200 – разрешена запись для пользователя, создавшего сегмент;
0040 – разрешено чтение для группы пользователя, создавшего сегмент;
0020 – разрешена запись для группы пользователя, создавшего сегмент;
0004 – разрешено чтение для всех остальных пользователей;
0002 – разрешена запись для всех остальных пользователей.

Возвращаемое значение

Системный вызов возвращает значение дескриптора System V IPC для сегмента разделяемой памяти при нормальном завершении и значение -1 при возникновении ошибки.

Доступ к созданной области разделяемой памяти в дальнейшем обеспечивается ее дескриптором, который вернет системный вызов shmget(). Доступ к уже существующей области также может осуществляться двумя способами:

- Если мы знаем ее ключ, то, используя вызов shmget(), можем получить ее дескриптор. В этом случае нельзя указывать в качестве составной части флагов флаг IPC_EXCL, а значение ключа, естественно, не может быть IPC_PRIVATE. Права доступа игнорируются, а размер области должен совпадать с размером, указанным при ее создании.
- Либо мы можем воспользоваться тем, что дескриптор System V IPC действителен в рамках всей операционной системы, и передать его значение от процесса, создавшего разделяемую память, текущему процессу. Отметим, что при создании разделяемой памяти с помощью значения IPC_PRIVATE – это единственно возможный способ.

После получения дескриптора необходимо включить область разделяемой памяти в адресное пространство текущего процесса. Это осуществляется с помощью системного вызова shmat(). При нормальном завершении он вернет адрес разделяемой памяти в адресном пространстве текущего процесса. Дальнейший доступ к этой памяти осуществляется с помощью обычных средств языка программирования.

Системный вызов shmat()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr, int shmflg);
```

Описание системного вызова

Системный вызов shmat предназначен для включения области разделяемой памяти в адресное пространство текущего процесса. Данное описание не является полным описанием системного вызова, а ограничивается рамками текущего курса. Для полного описания обращайтесь к UNIX Manual.

Параметр shmid является дескриптором System V IPC для сегмента разделяемой памяти, т. е. значением, которое вернул системный вызов shmget() при создании сегмента или при его поиске по ключу.

В качестве параметра shmaddr в рамках нашего курса мы всегда будем передавать значение NULL, позволяя операционной системе самой разместить разделяемую память в адресном пространстве нашего процесса.

Параметр shmflg в нашем курсе может принимать только два значения: 0 – для осуществления операций чтения и записи над сегментом и SHM_RDONLY – если мы хотим только читать из него. При этом процесс должен иметь соответствующие права доступа к сегменту.

Возвращаемое значение

Системный вызов возвращает адрес сегмента разделяемой памяти в адресном пространстве процесса при нормальном завершении и значение -1 при возникновении ошибки.

После окончания использования разделяемой памяти процесс может уменьшить размер своего адресного пространства, исключив из него эту область с помощью системного вызова shmdt(). Отметим, что в качестве параметра системный вызов shmdt() требует адрес начала области разделяемой памяти в адресном пространстве процесса, т. е. значение, которое вернул системный вызов shmat(), поэтому данное значение следует сохранять на протяжении всего времени использования разделяемой памяти.

Системный вызов shmdt()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(char *shmaddr);
```

Описание системного вызова

Системный вызов shmdt предназначен для исключения области разделяемой памяти из адресного пространства текущего процесса.

Параметр shmaddr является адресом сегмента разделяемой памяти, т. е. значением, которое вернул системный вызов shmat().

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Прогон программ с использованием разделяемой памяти

Для иллюстрации использования разделяемой памяти давайте рассмотрим две взаимодействующие программы:

```
/* Программа 1 (06-1a.c) для иллюстрации работы с
разделяемой памятью */
/* Мы организуем разделяемую память для массива из
трех целых чисел. Первый элемент массива является
счетчиком числа запусков программы 1, т. е. данной
программы, второй элемент массива - счетчиком числа
запусков программы 2, третий элемент массива -
счетчиком числа запусков обеих программ */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
{
  int *array; /* Указатель на разделяемую память */
```

```
int shmid; /* IPC дескриптор для области
  разделяемой памяти */
int new = 1; /* Флаг необходимости инициализации
  элементов массива */
char pathname[] = "06-1a.c"; /* Имя файла,
  используемое для генерации ключа. Файл с таким
  именем должен существовать в текущей директории */
key_t key; /* IPC ключ */
/* Генерируем IPC ключ из имени файла 06-1a.c в
  текущей директории и номера экземпляра области
  разделяемой памяти 0 */
if((key = ftok(pathname,0)) < 0){
  printf("Can\'t generate key\n");
  exit(-1);
}
/* Пытаемся эксклюзивно создать разделяемую память
для сгенерированного ключа, т. е. если для этого
ключа она уже существует, системный вызов вернет
отрицательное значение. Размер памяти определяем
как размер массива из трех целых переменных, права
доступа 0666 - чтение и запись разрешены для всех */
if((shmid = shmget(key, 3*sizeof(int),
0666|IPC_CREAT|IPC_EXCL)) < 0){
/* В случае ошибки пытаемся определить: возникла ли
она из-за того, что сегмент разделяемой памяти уже
существует или по другой причине */
  if(errno != EEXIST){
    /* Если по другой причине - прекращаем работу */
    printf("Can\'t create shared memory\n");
    exit(-1);
  } else {
    /* Если из-за того, что разделяемая память
    уже существует, то пытаемся получить ее IPC
    дескриптор и, в случае удачи, сбрасываем
    флаг необходимости инициализации элементов
    массива */
    if((shmid = shmget(key, 3*sizeof(int), 0)) < 0){
      printf("Can\'t find shared memory\n");
      exit(-1);
    }
    new = 0;
  }
```

```
    }
    /* Пытаемся отобразить разделяемую память в адресное
    пространство текущего процесса. Обратите внимание на
    то, что для правильного сравнения мы явно
    преобразовываем значение -1 к указателю на целое.*/
    if((array = (int *)shmat(shmid, NULL, 0)) == (int *)(-1)){
      printf("Can't attach shared memory\n");
      exit(-1);
    }
    /* В зависимости от значения флага new либо
    инициализируем массив, либо увеличиваем
    соответствующие счетчики */
    if(new){
      array[0] = 1;
      array[1] = 0;
      array[2] = 1;
    } else {
      array[0] += 1;
      array[2] += 1;
    }
    /* Печатаем новые значения счетчиков, удаляем
    разделяемую память из адресного пространства
    текущего процесса и завершаем работу */
    printf("Program 1 was spawn %d times,
      program 2 - %d times, total - %d times\n",
      array[0], array[1], array[2]);
    if(shmdt(array) < 0){
      printf("Can't detach shared memory\n");
      exit(-1);
    }
    return 0;
}

/* Программа 2 (06-1b.c) для иллюстрации работы с
разделяемой памятью */
/* Мы организуем разделяемую память для массива из
трех целых чисел. Первый элемент массива является
счетчиком числа запусков программы 1, т. е. данной
программы, второй элемент массива - счетчиком числа
запусков программы 2, третий элемент массива -
счетчиком числа запусков обеих программ */
#include <sys/types.h>
```

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
{
  int *array; /* Указатель на разделяемую память */
  int shmid; /* IPC дескриптор для области
    разделяемой памяти */
  int new = 1; /* Флаг необходимости инициализации
    элементов массива */
  char pathname[] = "06-1a.c"; /* Имя файла,
    используемое для генерации ключа. Файл с таким
    именем должен существовать в текущей директории */
  key_t key; /* IPC ключ */
  /* Генерируем IPC ключ из имени файла 06-1a.c в
  текущей директории и номера экземпляра области
  разделяемой памяти 0 */
  if((key = ftok(pathname,0)) < 0){
    printf("Can\'t generate key\n");
    exit(-1);
  }
  /* Пытаемся эксклюзивно создать разделяемую память
  для сгенерированного ключа, т.е. если для этого
  ключа она уже существует, системный вызов вернет
  отрицательное значение. Размер памяти определяем
  как размер массива из трех целых переменных, права
  доступа 0666 - чтение и запись разрешены для всех */
  if((shmid = shmget(key, 3*sizeof(int),
    0666|IPC_CREAT|IPC_EXCL)) < 0){
  /* В случае возникновения ошибки пытаемся определить:
  возникла ли она из-за того, что сегмент разделяемой
  памяти уже существует или по другой причине */
    if(errno != EEXIST){
      /* Если по другой причине - прекращаем работу */
      printf("Can\'t create shared memory\n");
      exit(-1);
    } else {
      /* Если из-за того, что разделяемая память
      уже существует, то пытаемся получить ее IPC
      дескриптор и, в случае удачи, сбрасываем флаг
      необходимости инициализации элементов массива */
```

```
            if((shmid = shmget(key, 3*sizeof(int), 0)) < 0){
               printf("Can\'t find shared memory\n");
               exit(-1);
            }
            new = 0;
         }
      }
      /* Пытаемся отобразить разделяемую память в адресное
      пространство текущего процесса. Обратите внимание
      на то, что для правильного сравнения мы явно
      преобразовываем значение -1 к указателю на целое.*/
      if((array = (int *)shmat(shmid, NULL, 0)) ==
         (int *)(-1)){
         printf("Can't attach shared memory\n");
         exit(-1);
      }
      /* В зависимости от значения флага new либо
      инициализируем массив, либо увеличиваем
      соответствующие счетчики */
      if(new){
         array[0] = 0;
         array[1] = 1;
         array[2] = 1;
      } else {
         array[1] += 1;
         array[2] += 1;
      }
      /* Печатаем новые значения счетчиков, удаляем
      разделяемую память из адресного пространства
      текущего процесса и завершаем работу */
      printf("Program 1 was spawn %d times,
         program 2 - %d times, total - %d times\n",
      array[0], array[1], array[2]);
      if(shmdt(array) < 0){
         printf("Can't detach shared memory\n");
         exit(-1);
      }
      return 0;
   }
```

Эти программы очень похожи друг на друга и используют разделяемую память для хранения числа запусков каждой из программ и их суммы.

В разделяемой памяти размещается массив из трех целых чисел. Первый элемент массива используется как счетчик для программы 1, второй элемент – для программы 2, третий элемент – для обеих программ суммарно. Дополнительный нюанс в программах возникает из-за необходимости инициализации элементов массива при создании разделяемой памяти. Для этого нам нужно, чтобы программы могли различать случай, когда они создали ее, и случай, когда она уже существовала. Мы добиваемся различия, используя вначале системный вызов `shmget()` с флагами `IPC_CREAT` и `IPC_EXCL`. Если вызов завершается нормально, то мы создали разделяемую память. Если вызов завершается с констатацией ошибки и значение переменной errno равняется `EEXIST`, то, значит, разделяемая память уже существует, и мы можем получить ее IPC-дескриптор, применяя тот же самый вызов с нулевым значением флагов. Наберите программы, сохраните под именами 06-1a.c и 06-1b.c соответственно, откомпилируйте их и запустите несколько раз. Проанализируйте полученные результаты.

Команды ipcs и ipcrm

Как мы видели из предыдущего примера, созданная область разделяемой памяти сохраняется в операционной системе даже тогда, когда нет ни одного процесса, включающего ее в свое адресное пространство. С одной стороны, это имеет определенные преимущества, поскольку не требует одновременного существования взаимодействующих процессов, с другой стороны, может причинять существенные неудобства. Допустим, что предыдущие программы мы хотим использовать таким образом, чтобы подсчитывать количество запусков в течение одного, текущего, сеанса работы в системе. Однако в созданном сегменте разделяемой памяти остается информация от предыдущего сеанса, и программы будут выдавать общее количество запусков за все время работы с момента загрузки операционной системы. Можно было бы создавать для нового сеанса новый сегмент разделяемой памяти, но количество ресурсов в системе не безгранично. Нас спасает то, что существуют способы удалять неиспользуемые ресурсы System V IPC как с помощью команд операционной системы, так и с помощью системных вызовов. Все средства System V IPC требуют определенных действий для освобождения занимаемых ресурсов после окончания взаимодействия процессов. Для того чтобы удалять ресурсы System V IPC из командной строки, нам понадобятся две команды, `ipcs` и `ipcrm`.

Команда `ipcs` выдает информацию обо всех средствах System V IPC, существующих в системе, для которых пользователь обладает правами на чтение: областях разделяемой памяти, семафорах и очередях сообщений.

Команда ipcs

Синтаксис команды

```
ipcs [-asmq] [-tclup]
ipcs [-smq] -i id
ipcs -h
```

Описание команды

Команда ipcs предназначена для получения информации о средствах System V IPC, к которым пользователь имеет право доступа на чтение.

Опция -i позволяет указать идентификатор ресурсов. Будет выдаваться только информация для ресурсов, имеющих этот идентификатор.

Виды IPC ресурсов могут быть заданы с помощью следующих опций:

- -s для семафоров;
- -m для сегментов разделяемой памяти;
- -q для очередей сообщений;
- -a для всех ресурсов (по умолчанию).

Опции [-tclup] используются для изменения состава выходной информации. По умолчанию для каждого средства выводятся его ключ, идентификатор IPC, идентификатор владельца, права доступа и ряд других характеристик. Применение опций позволяет вывести:

- -t времена совершения последних операций над средствами IPC;
- -p идентификаторы процесса, создавшего ресурс, и процесса, совершившего над ним последнюю операцию;
- -c идентификаторы пользователя и группы для создателя ресурса и его собственника;
- -l системные ограничения для средств System V IPC;
- -u общее состояние IPC ресурсов в системе.

Опция -h используется для получения краткой справочной информации.

Из всего многообразия выводимой информации нас будут интересовать только IPC идентификаторы для средств, созданных вами. Эти идентификаторы будут использоваться в команде ipcrm, позволяющей удалить необходимый ресурс из системы. Для удаления сегмента разделяемой памяти эта команда имеет вид

```
ipcrm shm <IPC идентификатор>
```

Удалите созданный вами сегмент разделяемой памяти из операционной системы, используя эти команды.

Команда ipcrm

Синтаксис команды

```
ipcrm [shm | msg | sem] id
```

Описание команды

Команда ipcrm предназначена для удаления ресурса System V IPC из операционной системы. Параметр id задает IPC-идентификатор для удаляемого ресурса, параметр shm используется для сегментов разделяемой памяти, параметр msg – для очередей сообщений, параметр sem – для семафоров.

Если поведение программ, использующих средства System V IPC, базируется на предположении, что эти средства были созданы при их работе, не забывайте перед их запуском удалять уже существующие ресурсы.

Использование системного вызова shmctl() для освобождения ресурса

Для той же цели – удалить область разделяемой памяти из системы – можно воспользоваться и системным вызовом shmctl(). Этот системный вызов позволяет полностью ликвидировать область разделяемой памяти в операционной системе по заданному дескриптору средства IPC, если, конечно, у вас хватает для этого полномочий. Системный вызов shmctl() позволяет выполнять и другие действия над сегментом разделяемой памяти, но их изучение лежит за пределами нашего курса.

Системный вызов shmctl()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Описание системного вызова

Системный вызов shmctl предназначен для получения информации об области разделяемой памяти, изменения ее атрибутов и удаления из системы. Данное описание не является полным описанием системного вызова, а ограничивается рамками текущего курса. Для изучения полного описания обращайтесь к UNIX Manual.

В нашем курсе мы будем пользоваться системным вызовом shmctl только для удаления области разделяемой памяти из системы. Параметр shmid является дескриптором System V IPC для сегмента разделяемой памяти, т. е. значением, которое вернул системный вызов shmget() при создании сегмента или при его поиске по ключу.

В качестве параметра cmd в рамках нашего курса мы всегда будем передавать значение IPC_RMID – команду для удаления сегмента разделяемой памяти с заданным идентификатором. Параметр buf для этой команды не используется, поэтому мы всегда будем подставлять туда значение NULL.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Разделяемая память и системные вызовы fork(), exec() и функция exit()

Важным вопросом является поведение сегментов разделяемой памяти при выполнении процессом системных вызовов fork(), exec() и функции exit().

При выполнении системного вызова fork() все области разделяемой памяти, размещенные в адресном пространстве процесса, наследуются порожденным процессом.

При выполнении системных вызовов exec() и функции exit() все области разделяемой памяти, размещенные в адресном пространстве процесса, исключаются из его адресного пространства, но продолжают существовать в операционной системе.

Самостоятельное написание, компиляция и запуск программы для организации связи двух процессов через разделяемую память

Для закрепления полученных знаний напишите две программы, осуществляющие взаимодействие через разделяемую память. Первая программа должна создавать сегмент разделяемой памяти и копировать туда собственный исходный текст, вторая программа должна брать оттуда этот текст, печатать его на экране и удалять сегмент разделяемой памяти из системы.

Понятие о нити исполнения (thread) в UNIX. Идентификатор нити исполнения. Функция pthread_self()

На лекции 4 мы говорили, что во многих современных операционных системах существует расширенная реализация понятия *процесс*, когда процесс представляет собой совокупность выделенных ему ресурсов и набора нитей исполнения. Нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет собственный программный счетчик, свое содержимое регистров и свой стек. Поскольку глобальные переменные у нитей исполнения являются общими, они могут использовать их как элементы разделяемой памяти, не прибегая к механизму, описанному выше.

В различных версиях операционной системы UNIX существуют различные интерфейсы, обеспечивающие работу с нитями исполнения. Мы кратко ознакомимся с некоторыми функциями, позволяющими разделить процесс на thread'ы и управлять их поведением, в соответствии со стандартом POSIX. Нити исполнения, удовлетворяющие стандарту POSIX, принято называть POSIX thread'ами или, кратко, pthread'ами.

К сожалению, операционная система Linux не полностью поддерживает нити исполнения на уровне ядра системы. При создании нового thread'a запускается новый традиционный процесс, разделяющий с родительским традиционным процессом его ресурсы, программный код и данные, расположенные вне стека, т. е. фактически действительно создается новый thread, но ядро не умеет определять, что эти thread'ы являются составными частями одного целого. Это «знает» только специальный процесс-координатор, работающий на пользовательском уровне и стартующий при первом вызове функций, обеспечивающих POSIX интерфейс для нитей исполнения. Поэтому мы сможем наблюдать не все преимущества использования нитей исполнения (в частности, ускорить решение задачи на однопроцессорной машине с их помощью вряд ли получится), но даже в этом случае thread'ы можно задействовать как очень удобный способ для создания процессов с общими ресурсами, программным кодом и разделяемой памятью.

Каждая нить исполнения, как и процесс, имеет в системе уникальный номер – идентификатор thread'a. Поскольку традиционный процесс в концепции нитей исполнения трактуется как процесс, содержащий единственную нить исполнения, мы можем узнать идентификатор этой нити и для любого обычного процесса. Для этого используется функция `pthread_self()`. Нить исполнения, создаваемую при рождении нового процесса, принято называть *начальной* или *главной нитью исполнения* этого процесса.

Функция pthread_self()

Прототип функции

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Описание функции

Функция `pthread_self` возвращает идентификатор текущей нити исполнения.
Тип данных `pthread_t` является синонимом для одного из целочисленных типов языка С.

Создание и завершение thread'а. Функции pthread_create(), pthread_exit(), pthread_join()

Нити исполнения, как и традиционные процессы, могут порождать нити-потомки, правда, только внутри своего процесса. Каждый будущий thread внутри программы должен представлять собой функцию с прототипом

```
void *thread(void *arg);
```

Параметр `arg` передается этой функции при создании thread'а и может, до некоторой степени, рассматриваться как аналог параметров функции `main()`, о которых мы говорили на семинарах 3–4. Возвращаемое функцией значение может интерпретироваться как аналог информации, которую родительский процесс может получить после завершения процесса-ребенка. Для создания новой нити исполнения применяется функция `pthread_create()`.

Функция для создания нити исполнения

Прототип функции

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
   void * (*start_routine)(void *), void *arg);
```

Описание функции

Функция `pthread_create` служит для создания новой нити исполнения (thread'а) внутри текущего процесса. Настоящее описание не является полным описанием функции, а служит только целям данного курса. Для изучения полного описания обращайтесь к UNIX Manual.

Новый thread будет выполнять функцию start_routine с прототипом

```
void *start_routine(void *)
```

передавая ей в качестве аргумента параметр arg. Если требуется передать более одного параметра, они собираются в структуру, и передается адрес этой структуры. Значение, возвращаемое функцией start_routine, не должно указывать на динамический объект данного thread'a.

Параметр attr служит для задания различных атрибутов создаваемого thread'a. Их описание выходит за рамки нашего курса, и мы всегда будем считать их заданными по умолчанию, подставляя в качестве аргумента значение NULL.

Возвращаемые значения

При удачном завершении функция возвращает значение 0 и помещает идентификатор новой нити исполнения по адресу, на который указывает параметр thread. В случае ошибки возвращается **положительное значение** (а не отрицательное, как в большинстве системных вызовов и функций!), которое определяет код ошибки, описанный в файле <errno.h>. Значение системной переменной errno при этом не устанавливается.

Мы не будем рассматривать ее в полном объеме, так как детальное изучение программирования с использованием thread'ов не является целью данного курса.

Важным отличием этой функции от большинства других системных вызовов и функций является то, что в случае неудачного завершения она **возвращает не отрицательное, а положительное значение**, которое определяет код ошибки, описанный в файле <errno.h>. Значение системной переменной errno при этом не устанавливается. Результатом выполнения этой функции является появление в системе новой нити исполнения, которая будет выполнять функцию, ассоциированную со thread'ом, передав ей специфицированный параметр, параллельно с уже существовавшими нитями исполнения процесса.

Созданный thread может завершить свою деятельность тремя способами:

- с помощью выполнения функции pthread_exit(). Функция никогда не возвращается в вызвавшую ее нить исполнения. Объект, на который указывает параметр этой функции, может быть изучен в другой нити исполнения, например, в породившей завершившийся thread. Этот параметр, следовательно, должен указывать на объект, не являющийся локальным для завершившегося thread'a, например, на статическую переменную;
- с помощью возврата из функции, ассоциированной с нитью исполнения. Объект, на который указывает адрес, возвращаемый функцией, как и в предыдущем случае, может быть изучен в другой нити

исполнения, например, в породившей завершившийся thread, и должен указывать на объект, не являющийся локальным для завершившегося thread'a;

- если в процессе выполняется возврат из функции `main()` или где-либо в процессе (в любой нити исполнения) осуществляется вызов функции `exit()`, это приводит к завершению всех thread'ов процесса.

Функция для завершения нити исполнения

Прототип функции

```
#include <pthread.h>
void pthread_exit(void *status);
```

Описание функции

Функция `pthread_exit` служит для завершения нити исполнения (thread) текущего процесса.

Функция никогда не возвращается в вызвавший ее thread. Объект, на который указывает параметр `status`, может быть впоследствии изучен в другой нити исполнения, например в нити, породившей завершившуюся нить. Поэтому он не должен указывать на динамический объект завершившегося thread'a.

Одним из вариантов получения адреса, возвращаемого завершившимся thread'ом, с одновременным ожиданием его завершения является использование функции `pthread_join()`. Нить исполнения, вызвавшая эту функцию, переходит в состояние ***ожидание*** до завершения заданного thread'a. Функция позволяет также получить указатель, который вернул завершившийся thread в операционную систему.

Функция pthread_join()

Прототип функции

```
#include <pthread.h>
int pthread_join (pthread_t thread,
void **status_addr);
```

Описание функции

Функция `pthread_join` блокирует работу вызвавшей ее нити исполнения до завершения thread'a с идентификатором `thread`. После разблокирования в указатель, расположенный по адресу `status_addr`, заносится адрес, который вернул завершившийся thread либо при выходе из ассоциированной с ним функции, либо при выполнении функции `pthread_exit()`. Если

нас не интересует, что вернула нам нить исполнения, в качестве этого параметра можно использовать значение NULL.

Возвращаемые значения

Функция возвращает значение 0 при успешном завершении. В случае ошибки возвращается **положительное значение** (а не отрицательное, как в большинстве системных вызовов и функций!), которое определяет код ошибки, описанный в файле <errno.h>. Значение системной переменной errno при этом не устанавливается.

Прогон программы с использованием двух нитей исполнения

Для иллюстрации вышесказанного давайте рассмотрим программу, в которой работают две нити исполнения:

```
/* Программа 06-2.c для иллюстрации работы двух нитей
исполнения. Каждая нить исполнения просто увеличивает
на 1 разделяемую переменную a. */
#include <pthread.h>
#include <stdio.h>
int a = 0;
/* Переменная a является глобальной статической для
всей программы, поэтому она будет разделяться обеими
нитями исполнения.*/
/* Ниже следует текст функции, которая будет
ассоциирована со 2-м thread'ом */
void *mythread(void *dummy)
/* Параметр dummy в нашей функции не используется и
присутствует только для совместимости типов данных.
По той же причине функция возвращает значение
void *, хотя это никак не используется в программе.*/
{
   pthread_t mythid; /* Для идентификатора нити
   исполнения */
   /* Заметим, что переменная mythid является
   динамической локальной переменной функции
   mythread(), т. е. помещается в стеке и,
   следовательно, не разделяется нитями исполнения. */
   /* Запрашиваем идентификатор thread'a */
   mythid = pthread_self();
   a = a+1;
```

```
    printf("Thread %d, Calculation result = %d\n",
      mythid, a);
    return NULL;
}
/* Функция main() - она же ассоциированная функция
главного thread'a */
int main()
{
    pthread_t thid, mythid;
    int result;
    /* Пытаемся создать новую нить исполнения,
    ассоциированную с функцией mythread(). Передаем ей
    в качестве параметра значение NULL. В случае удачи в
    переменную thid занесется идентификатор нового
    thread'a. Если возникнет ошибка, то прекратим
    работу. */
    result = pthread_create( &thid,
      (pthread_attr_t *)NULL, mythread, NULL);
    if(result != 0){
    printf ("Error on thread create,
      return value = %d\n", result);
    exit(-1);
    }
    printf("Thread created, thid = %d\n", thid);
    /* Запрашиваем идентификатор главного thread'a */
    mythid = pthread_self();
    a = a+1;
    printf("Thread %d, Calculation result = %d\n",
      mythid, a);
    /* Ожидаем завершения порожденного thread'a,
    не интересуясь, какое значение он нам вернет. Если
    не выполнить вызов этой функции, то возможна ситуа-
    ция, когда мы завершим функцию main() до того,
    как выполнится порожденный thread, что автомати-
    чески повлечет за собой его завершение, исказив
    результаты. */
    pthread_join(thid, (void **)NULL);
    return 0;
}
```

Для сборки исполняемого файла при работе редактора связей необходимо явно подключить библиотеку функций для работы с pthread'ами,

которая не подключается автоматически. Это делается с помощью добавления к команде компиляции и редактирования связей параметра **-lpthread** – подключить библиотеку pthread. Наберите текст, откомпилируйте эту программу и запустите на исполнение.

Обратите внимание на отличие результатов этой программы от похожей программы, иллюстрировавшей создание нового процесса (раздел «Прогон программы с fork() с одинаковой работой родителя и ребенка»), которую мы рассматривали на семинарах 3 – 4. Программа, создававшая новый процесс, печатала дважды одинаковые значения для переменной a, так как адресные пространства различных процессов независимы, и каждый процесс прибавлял 1 к своей собственной переменной a. Рассматриваемая программа печатает два разных значения, так как переменная a является разделяемой, и каждый thread прибавляет 1 к одной и той же переменной.

Написание, компиляция и прогон программы с использованием трех нитей исполнения

Модифицируйте предыдущую программу, добавив к ней третью нить исполнения.

Необходимость синхронизации процессов и нитей исполнения, использующих общую память

Все рассмотренные на этом семинаре примеры являются не совсем корректными. В большинстве случаев они работают правильно, однако возможны ситуации, когда совместная деятельность этих процессов или нитей исполнения приводит к неверным и неожиданным результатам. Это связано с тем, что любые неатомарные операции, связанные с изменением содержимого разделяемой памяти, представляют собой критическую секцию процесса или нити исполнения. Вспомните рассмотрение критических секций в лекции 5.

Вернемся к рассмотрению программ из раздела «Прогон программ с использованием разделяемой памяти». При одновременном существовании двух процессов в операционной системе может возникнуть следующая последовательность выполнения операций во времени:

```
...
Процесс 1: array[0] += 1;
Процесс 2: array[1] += 1;
Процесс 1: array[2] += 1;
Процесс 1: printf("Program 1 was spawn %d times,
```

```
    program 2 - %d times, total - %d times\n",
    array[0], array[1], array[2]);
...
```

Тогда печать будет давать неправильные результаты. Естественно, что воспроизвести подобную последовательность действий практически нереально. Мы не сможем подобрать необходимое время старта процессов и степень загруженности вычислительной системы. Но мы можем смоделировать эту ситуацию, добавив в обе программы достаточно длительные пустые циклы перед оператором `array[2] + = 1;` Это проделано в следующих программах:

```
/* Программа 1 (06-3a.c) для иллюстрации
некорректной работы с разделяемой памятью */
/* Мы организуем разделяемую память для массива из
трех целых чисел. Первый элемент массива является
счетчиком числа запусков программы 1, т. е. данной
программы, второй элемент массива - счетчиком числа
запусков программы 2, третий элемент массива -
счетчиком числа запусков обеих программ */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
{
  int *array; /* Указатель на разделяемую память */
  int shmid; /* IPC-дескриптор для области
  разделяемой памяти */
  int new = 1; /* Флаг необходимости инициализации
    элементов массива */
  char pathname[] = "06-3a.c"; /* Имя файла,
    использующееся для генерации ключа. Файл с таким
    именем должен существовать в текущей директории */
  key_t key; /* IPC-ключ */
  long i;
  /* Генерируем IPC-ключ из имени файла 06-3a.c в
  текущей директории и номера экземпляра области
  разделяемой памяти 0 */
  if((key = ftok(pathname,0)) < 0){
    printf("Can\'t generate key\n");
```

```
    exit(-1);
  }
  /* Пытаемся эксклюзивно создать разделяемую память
  для сгенерированного ключа, т. е. если для этого
  ключа она уже существует, системный вызов вернет
  отрицательное значение. Размер памяти определяем
  как размер массива из трех целых переменных, права
  доступа 0666 - чтение и запись разрешены для всех */
  if((shmid = shmget(key, 3*sizeof(int),
    0666|IPC_CREAT|IPC_EXCL)) < 0){
  /* В случае возникновения ошибки пытаемся определить:
  возникла ли она из-за того, что сегмент разделяемой
  памяти уже существует или по другой причине */
    if(errno != EEXIST){
      /* Если по другой причине - прекращаем работу */
      printf("Can\'t create shared memory\n");
      exit(-1);
    } else {
      /* Если из-за того, что разделяемая память
      уже существует - пытаемся получить ее IPC-
      дескриптор и, в случае удачи, сбрасываем
      флаг необходимости инициализации элементов
      массива */
      if((shmid = shmget(key, 3*sizeof(int), 0)) < 0){
        printf("Can\'t find shared memory\n");
        exit(-1);
      }
      new = 0;
    }
  }
  /* Пытаемся отобразить разделяемую память в адресное
  пространство текущего процесса. Обратите внимание
  на то, что для правильного сравнения мы явно
  преобразовываем значение -1 к указателю на целое.*/
  if((array = (int *)shmat(shmid, NULL, 0)) ==
    (int *)(-1)){
    printf("Can't attach shared memory\n");
    exit(-1);
  }
/* В зависимости от значения флага new либо
инициализируем массив, либо увеличиваем
соответствующие счетчики */
```

```
  if(new){
    array[0] = 1;
    array[1] = 0;
    array[2] = 1;
  } else {
    array[0] += 1;
    for(i=0; i<1000000000L; i++);
    /* Предельное значение для i может меняться в
    зависимости от производительности компьютера */
    array[2] += 1;
  }
  /* Печатаем новые значения счетчиков, удаляем
  разделяемую память из адресного пространства
  текущего процесса и завершаем работу */
  printf("Program 1 was spawn %d times,
    program 2 - %d times, total - %d times\n",
    array[0], array[1], array[2]);
  if(shmdt(array) < 0){
    printf("Can't detach shared memory\n");
    exit(-1);
  }
  return 0;
}

/* Программа 2 (06-3b.c) для иллюстрации
некорректной работы с разделяемой памятью */
/* Мы организуем разделяемую память для массива
из трех целых чисел. Первый элемент массива
является счетчиком числа запусков программы 1,
т. е. данной программы, второй элемент массива -
счетчиком числа запусков программы 2, третий
элемент массива - счетчиком числа запусков обеих
программ */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
{
int *array; /* Указатель на разделяемую память */
  int shmid; /* IPC-дескриптор для области
```

```
    разделяемой памяти */
  int new = 1; /* Флаг необходимости инициализации
    элементов массива */
  char pathname[] = "06-3a.c"; /* Имя файла,
    использующееся для генерации ключа. Файл с таким
    именем должен существовать в текущей директории */
  key_t key; /* IPC-ключ */
  long i;
  /* Генерируем IPC-ключ из имени файла 06-3a.c в
  текущей директории и номера экземпляра области
  разделяемой памяти 0 */
  if((key = ftok(pathname,0)) < 0){
    printf("Can\'t generate key\n");
    exit(-1);
  }
  /* Пытаемся эксклюзивно создать разделяемую память
  для сгенерированного ключа, т. е. если для этого
  ключа она уже существует, системный вызов вернет
  отрицательное значение. Размер памяти определяем
  как размер массива из трех целых переменных, права
  доступа 0666 - чтение и запись разрешены для всех */
  if((shmid = shmget(key, 3*sizeof(int),
    0666|IPC_CREAT|IPC_EXCL)) < 0){
  /* В случае ошибки пытаемся определить, возникла
  ли она из-за того, что сегмент разделяемой памяти
  уже существует или по другой причине */
    if(errno != EEXIST){
      /* Если по другой причине - прекращаем работу */
      printf("Can\'t create shared memory\n");
      exit(-1);
    } else {
      /* Если из-за того, что разделяемая память уже
      существует - пытаемся получить ее IPC-дескрип-
      тор и, в случае удачи, сбрасываем флаг
      необходимости инициализации элементов массива */
      if((shmid = shmget(key,
        3*sizeof(int), 0)) < 0){
        printf("Can\'t find shared memory\n");
        exit(-1);
      }
      new = 0;
    }
```

```
    }
    /* Пытаемся отобразить разделяемую память в адрес-
    ное пространство текущего процесса. Обратите внима-
    ние на то, что для правильного сравнения мы явно
    преобразовываем значение -1 к указателю на целое.*/
    if((array = (int *)shmat(shmid, NULL, 0)) ==
      (int *)(-1)){
      printf("Can't attach shared memory\n");
      exit(-1);
    }
    /* В зависимости от значения флага new либо
    инициализируем массив, либо увеличиваем
    соответствующие счетчики */
    if(new){
      array[0] = 0;
      array[1] = 1;
      array[2] = 1;
    } else {
      array[1] += 1;
      for(i=0; i<1000000000L; i++);
      /* Предельное значение для i может меняться в
      зависимости от производительности компьютера */
      array[2] += 1;
    }
    /* Печатаем новые значения счетчиков, удаляем
    разделяемую память из адресного пространства
    текущего процесса и завершаем работу */
    printf("Program 1 was spawn %d times,
      program 2 - %d times, total - %d times\n",
      array[0], array[1], array[2]);
      if(shmdt(array) < 0){
        printf("Can't detach shared memory\n");
        exit(-1);
      }
      return 0;
}
```

Наберите программы, сохраните под именами 06-3a.c и 06-3b.c соответственно, откомпилируйте их и запустите любую из них один раз для создания и инициализации разделяемой памяти. Затем запустите другую и, пока она находится в цикле, запустите (например, с другого виртуального терминала) снова первую программу. Вы получите неожиданный результат: ко-

личество запусков по отдельности не будет соответствовать количеству запусков вместе.

Как мы видим, для написания корректно работающих программ необходимо обеспечивать взаимоисключение при работе с разделяемой памятью и, может быть, взаимную очередность доступа к ней. Это можно сделать с помощью рассмотренных в лекции 6 алгоритмов синхронизации, например, алгоритма Петерсона или алгоритма булочной.

Задача повышенной сложности: модифицируйте программы из этого раздела для корректной работы с помощью алгоритма Петерсона.

На следующем семинаре мы рассмотрим семафоры, которые являются средством System V IPC, предназначенным для синхронизации процессов.

Семинар 8. Семафоры в UNIX как средство синхронизации процессов

Семафоры в UNIX. Отличие операций над UNIX-семафорами от классических операций. Создание массива семафоров или доступ к уже существующему массиву. Системный вызов semget(). Выполнение операций над семафорами. Системный вызов semop(). Удаление набора семафоров из системы с помощью команды ipcrm или системного вызова semctl(). Понятие о POSIX-семафорах.

Ключевые слова: семафоры System V IPC, массив семафоров, операция A(S, n), операция D(S, n), операция Z(S), системные вызовы semget(), semop(), semctl(), POSIX-семафоры.

Семафоры в UNIX. Отличие операций над UNIX-семафорами от классических операций

В материалах предыдущего семинара речь шла о необходимости синхронизации работы процессов для их корректного взаимодействия через разделяемую память. Как упоминалось в лекции 6, одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал Дейкстра (Dijkstra) в 1965 году. При разработке средств System V IPC семафоры вошли в их состав как неотъемлемая часть. Следует отметить, что набор операций над семафорами System V IPC отличается от классического набора операций {P, V}, предложенного Дейкстрой. Он включает три операции:

- A(S, n) – увеличить значение семафора S на величину n;
- D(S, n) – пока значение семафора S < n, процесс блокируется. Далее
  S = S - n;
- Z(S) – процесс блокируется до тех пор, пока значение семафора S не станет равным 0.

Изначально все IPC-семафоры инициируются нулевым значением.

Мы видим, что классической операции P(S) соответствует операция D(S,1), а классической операции V(S) соответствует операция A(S,1). Аналогом ненулевой инициализации семафоров Дейкстры значением n может служить выполнение операции A(S,n) сразу после создания семафора S, с обеспечением атомарности создания семафора и ее выполнения посредством другого семафора. Мы показали, что классические семафоры реализуются через семафоры System V IPC. Обратное не является верным. Используя операции P(S) и V(S), мы не сумеем реализовать операцию Z(S).

Поскольку IPC-семафоры являются составной частью средств System V IPC, то для них верно все, что говорилось об этих средствах в материалах предыдущего семинара. IPC-семафоры являются средством связи с непрямой адресацией, требуют инициализации для организации взаимодействия процессов и специальных действий для освобождения системных ресурсов по его окончании. Пространством имен IPC-семафоров является множество значений ключа, генерируемых с помощью функции `ftok()`. Для совершения операций над семафорами системным вызовам в качестве параметра передаются IPC-дескрипторы семафоров, однозначно идентифицирующих их во всей вычислительной системе, а вся информация о семафорах располагается в адресном пространстве ядра операционной системы. Это позволяет организовывать через семафоры взаимодействие процессов, даже не находящихся в системе одновременно.

Создание массива семафоров или доступ к уже существующему. Системный вызов semget()

В целях экономии системных ресурсов операционная система UNIX позволяет создавать не по одному семафору для каждого конкретного значения ключа, а связывать с ключом целый массив семафоров (в Linux – до 500 семафоров в массиве, хотя это количество может быть уменьшено системным администратором). Для создания массива семафоров, ассоциированного с определенным ключом, или доступа по ключу к уже существующему массиву используется системный вызов `semget()`, являющийся аналогом системного вызова `shmget()` для разделяемой памяти, который возвращает значение IPC-дескриптора для этого массива. При этом применяются те же способы создания и доступа (см. семинары 6–7 раздел «Разделяемая память в UNIX. Системные вызовы `shmget()`, `shmat()`, `shmdt()`»), что и для разделяемой памяти. Вновь созданные семафоры инициируются нулевым значением.

Системный вызов semget()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

Описание системного вызова

Системный вызов semget предназначен для выполнения операции доступа к массиву IPC-семафоров и, в случае ее успешного завершения, возвращает дескриптор System V IPC для этого массива (целое неотрицательное число, однозначно характеризующее массив семафоров внутри вычислительной системы и использующееся в дальнейшем для других операций с ним).

Параметр key является ключом System V IPC для массива семафоров, т. е. фактически его именем из пространства имен System V IPC. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции ftok(), или специальное значение IPC_PRIVATE. Использование значения IPC_PRIVATE **всегда** приводит к попытке создания нового массива семафоров с ключом, который не совпадает со значением ключа ни одного из уже существующих массивов и не может быть получен с помощью функции ftok() ни при одной комбинации ее параметров.

Параметр nsems определяет количество семафоров в создаваемом или уже существующем массиве. В случае, если массив с указанным ключом уже имеется, но его размер не совпадает с указанным в параметре nsems, констатируется возникновение ошибки.

Параметр semflg – флаги – играет роль только при создании нового массива семафоров и определяет права различных пользователей при доступе к массиву, а также необходимость создания нового массива и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или – «|») следующих предопределенных значений и восьмеричных прав доступа:

IPC_CREAT – если массива для указанного ключа не существует, он должен быть создан;

IPC_EXCL – применяется совместно с флагом IPC_CREAT. При совместном их использовании и существовании массива с указанным ключом, доступ к массиву не производится и констатируется ошибка, при этом переменная errno, описанная в файле <errno.h>, примет значение EEXIST

0400 – разрешено чтение для пользователя, создавшего массив;

0200 – разрешена запись для пользователя, создавшего массив;

0040 – разрешено чтение для группы пользователя, создавшего массив;

0020 – разрешена запись для группы пользователя, создавшего массив;

0004 – разрешено чтение для всех остальных пользователей;

0002 – разрешена запись для всех остальных пользователей.

Вновь созданные семафоры инициируются нулевым значением.

Возвращаемое значение

Системный вызов возвращает значение дескриптора System V IPC для массива семафоров при нормальном завершении и значение -1 при возникновении ошибки.

Выполнение операций над семафорами. Системный вызов semop()

Для выполнения операций A, D и Z над семафорами из массива используется системный вызов semop(), обладающий довольно сложной

семантикой. Разработчики System V IPC явно перегрузили этот вызов, применяя его не только для выполнения всех трех операций, но еще и для нескольких семафоров в массиве IPC-семафоров одновременно. Для правильного использования этого вызова необходимо выполнить следующие действия:

1. Определиться, для каких семафоров из массива предстоит выполнить операции. Необходимо иметь в виду, что все операции реально совершаются только перед успешным возвращением из системного вызова, т. е. если вы хотите выполнить операции A(S_1,5) и Z(S_2) в одном вызове и оказалось, что S_2 != 0, то значение семафора S_1 не будет изменено до тех пор, пока значение S_2 не станет равным 0. Порядок выполнения операций в случае, когда процесс не переходит в состояние ***ожидание***, не определен. Так, например, при одновременном выполнении операций A(S_1,1) и D(S_2,1) в случае $S_2 > 1$ неизвестно, что произойдет раньше – уменьшится значение семафора S_2 или увеличится значение семафора S_1. Если порядок для вас важен, лучше применить несколько вызовов вместо одного.
2. После того как вы определились с количеством семафоров и совершаемыми операциями, необходимо завести в программе массив из элементов типа `struct sembuf` с размерностью, равной определенному количеству семафоров (если операция совершается только над одним семафором, можно, естественно, обойтись просто переменной). Каждый элемент этого массива будет соответствовать операции над одним семафором.
3. Заполнить элементы массива. В поле `sem_flg` каждого элемента нужно занести значение 0 (другие значения флагов в семинарах мы рассматривать не будем). В поля `sem_num` и `sem_op` следует занести номера семафоров в массиве IPC семафоров и соответствующие коды операций. Семафоры нумеруются, начиная с 0. Если у вас в массиве всего один семафор, то он будет иметь номер 0. Операции кодируются так:
   – для выполнения операции A(S,n) значение поля `sem_op` должно быть равно n;
   – для выполнения операции D(S,n) значение поля `sem_op` должно быть равно -n;
   – для выполнения операции Z(S) значение поля `sem_op` должно быть равно 0.
4. В качестве второго параметра системного вызова `semop()` указать адрес заполненного массива, а в качестве третьего параметра – ранее определенное количество семафоров, над которыми совершаются операции.

Системный вызов semop()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, int nsops);
```

Описание системного вызова

Системный вызов semop предназначен для выполнения операций A, D и Z (см. описание операций над семафорами из массива IPC семафоров – раздел «Создание массива семафоров или доступ к уже существующему. Системный вызов semget()» этого семинара). Данное описание не является полным описанием системного вызова, а ограничивается рамками текущего курса. Для полного описания обращайтесь к UNIX Manual.

Параметр semid является дескриптором System V IPC для набора семафоров, т. е. значением, которое вернул системный вызов semget() при создании набора семафоров или при его поиске по ключу.

Каждый из nsops элементов массива, на который указывает параметр sops, определяет операцию, которая должна быть совершена над каким-либо семафором из массива IPC семафоров, и имеет тип структуры struct sembuf, в которую входят следующие переменные:

short sem_num – номер семафора в массиве IPC-семафоров (нумеруются, начиная с 0);

short sem_op – выполняемая операция;

short sem_flg – флаги для выполнения операции. В нашем курсе всегда будем считать эту переменную равной 0.

Значение элемента структуры sem_op определяется следующим образом:

– для выполнения операции A(S,n) значение должно быть равно n;

– для выполнения операции D(S,n) значение должно быть равно -n;

– для выполнения операции Z(S) значение должно быть равно 0.

Семантика системного вызова подразумевает, что все операции будут в реальности выполнены над семафорами только перед успешным возвращением из системного вызова. Если при выполнении операций D или Z процесс перешел в состояние ожидания, то он может быть выведен из этого состояния при возникновении следующих форс-мажорных ситуаций:

– массив семафоров был удален из системы;

– процесс получил сигнал, который должен быть обработан.

В этом случае происходит возврат из системного вызова с констатацией ошибочной ситуации.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Прогон примера с использованием семафора

Для иллюстрации сказанного рассмотрим простейшие программы, синхронизирующие свои действия с помощью семафоров:

```
/* Программа 08-1a.c для иллюстрации работы с
семафорами */
/* Эта программа получает доступ к одному системному
семафору, ждет, пока его значение не станет больше
или равным 1 после запусков программы 08-1b.c, а за-
тем уменьшает его на 1*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
int main()
{
  int semid; /* IPC-дескриптор для массива IPC
  семафоров */
  char pathname[] = "08-1a.c"; /* Имя файла,
    использующееся для генерации ключа. Файл с таким
    именем должен существовать в текущей директории */
  key_t key; /* IPC ключ */
  struct sembuf mybuf; /* Структура для задания
    операции над семафором */
  /* Генерируем IPC-ключ из имени файла 08-1a.c в
  текущей директории и номера экземпляра массива
  семафоров 0 */
  if((key = ftok(pathname,0)) < 0){
    printf("Can\'t generate key\n");
    exit(-1);
  }
  /* Пытаемся получить доступ по ключу к массиву
  семафоров, если он существует, или создать его из
  одного семафора, если его еще не существует, с правами
  доступа read & write для всех пользователей */
  if((semid = semget(key, 1, 0666 | IPC_CREAT)) < 0){
    printf("Can\'t get semid\n");
    exit(-1);
  }
  /* Выполним операцию D(semid1,1) для нашего массива
  семафоров. Для этого сначала заполним нашу структуру.
```

```
    Флаг, как обычно, полагаем равным 0. Наш массив
    семафоров состоит из одного семафора с номером 0.
    Код операции -1.*/
    mybuf.sem_op = -1;
    mybuf.sem_flg = 0;
    mybuf.sem_num = 0;
    if(semop(semid, &mybuf, 1) < 0){
      printf("Can\'t wait for condition\n");
      exit(-1);
    }
    printf("Condition is present\n");
    return 0;
}
/* Программа 08-1b.c для иллюстрации работы с
семафорами */
/* Эта программа получает доступ к одному системному
семафору и увеличивает его на 1*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
int main()
{
    int semid; /* IPC-дескриптор для массива IPC
      семафоров */
    char pathname[] = "08-1a.c"; /* Имя файла,
      использующееся для генерации ключа. Файл с таким
      именем должен существовать в текущей директории */
    key_t key; /* IPC ключ */
    struct sembuf mybuf; /* Структура для задания
    операции над семафором */
    /* Генерируем IPC-ключ из имени файла 08-1a.c в
    текущей директории и номера экземпляра массива
    семафоров 0 */
    if((key = ftok(pathname,0)) < 0){
      printf("Can\'t generate key\n");
      exit(-1);
    }
    /* Пытаемся получить доступ по ключу к массиву
    семафоров, если он существует, или создать его из
    одного семафора, если его еще не существует, с правами
    доступа read & write для всех пользователей */
```

```
    if((semid = semget(key, 1, 0666 | IPC_CREAT)) < 0){
      printf("Can\'t get semid\n");
      exit(-1);
    }
    /* Выполним операцию A(semid1,1) для нашего массива
    семафоров. Для этого сначала заполним нашу структуру.
    Флаг, как обычно, полагаем равным 0. Наш массив
    семафоров состоит из одного семафора с номером 0.
    Код операции 1.*/
    mybuf.sem_op = 1;
    mybuf.sem_flg = 0;
    mybuf.sem_num = 0;
    if(semop(semid, &mybuf, 1) < 0){
      printf("Can\'t wait for condition\n");
      exit(-1);
    }
    printf("Condition is set\n");
    return 0;
}
```

Первая программа выполняет над семафором `S` операцию `D(S,1)`, вторая программа выполняет над тем же семафором операцию `A(S,1)`. Если семафора в системе не существует, любая программа создает его перед выполнением операции. Поскольку при создании семафор всегда инициируется `0`, то программа 1 может работать без блокировки только после запуска программы 2. Наберите программы, сохраните под именами 08-1a.c и 08-1b.c соответственно, откомпилируйте и проверьте правильность их поведения.

Изменение предыдущего примера

Измените программы из предыдущего раздела так, чтобы первая программа могла работать без блокировки после не менее пяти запусков второй программы.

Удаление набора семафоров из системы с помощью команды ipcrm или системного вызова semctl()

Как мы видели в примерах, массив семафоров может продолжать существовать в системе и после завершения использовавших его процессов, а семафоры будут сохранять свое значение. Это может привести к некорректному поведению программ, предполагающих, что семафоры были

только что созданы и, следовательно, имеют нулевое значение. Необходимо удалять семафоры из системы перед запуском таких программ или перед их завершением. Для удаления семафоров можно воспользоваться командами `ipcs` и `ipcrm`, рассмотренными в материалах предыдущего семинара. Команда `ipcrm` в этом случае должна иметь вид

```
ipcrm sem <IPC идентификатор>
```

Для этой же цели мы можем применять системный вызов `semctl()`, который умеет выполнять и другие операции над массивом семафоров, но их рассмотрение выходит за рамки нашего курса.

Системный вызов semctl()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, union semun arg);
```

Описание системного вызова

Системный вызов `semctl` предназначен для получения информации о массиве IPC семафоров, изменения его атрибутов и удаления его из системы. Данное описание не является полным описанием системного вызова, а ограничивается рамками текущего курса. Для изучения полного описания обращайтесь к UNIX Manual.

В нашем курсе мы будем применять системный вызов `semctl` только для удаления массива семафоров из системы. Параметр `semid` является дескриптором System V IPC для массива семафоров, т. е. значением, которое вернул системный вызов `semget()` при создании массива или при его поиске по ключу.

В качестве параметра `cmd` в рамках нашего курса мы всегда будем передавать значение `IPC_RMID` – команду для удаления сегмента разделяемой памяти с заданным идентификатором. Параметры `semnum` и `arg` для этой команды не используются, поэтому мы всегда будем подставлять вместо них значение `0`.

Если какие-либо процессы находились в состоянии **ожидание** для семафоров из удаляемого массива при выполнении системного вызова `semop()`, то они будут разблокированы и вернутся из вызова `semop()` с индикацией ошибки.

Возвращаемое значение

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

Написание, компиляция и прогон программы с организацией взаимоисключения с помощью семафоров для двух процессов, взаимодействующих через разделяемую память

В материалах семинаров 6–7 было показано, что любые неатомарные операции, связанные с изменением содержимого разделяемой памяти, представляют собой критическую секцию процесса или нити исполнения. Модифицируйте программы из раздела «Необходимость синхронизации процессов и нитей исполнения, использующих общую память» семинаров 6–7, которые иллюстрировали некорректную работу через разделяемую память, обеспечив с помощью семафоров взаимоисключения для их правильной работы.

Написание, компиляция и прогон программы с организацией взаимной очередности с помощью семафоров для двух процессов, взаимодействующих через pipe

В материалах семинара 5, когда речь шла о связи родственных процессов через pipe, отмечалось, что pipe является однонаправленным каналом связи, и что для организации связи через один pipe в двух направлениях необходимо использовать механизмы взаимной синхронизации процессов. Организуйте двустороннюю поочередную связь процесса-родителя и процесса-ребенка через pipe, используя для синхронизации семафоры, модифицировав программу из раздела «Прогон программы для организации однонаправленной связи между родственными процессами через pipe» семинара 5.

Понятие о POSIX-семафорах

В стандарте POSIX вводятся другие семафоры, полностью аналогичные семафорам Дейкстры. Для инициализации значения таких семафоров применяется функция `sem_init()`, аналогом операции `P` служит функция `sem_wait()`, а аналогом операции `V` – функция `sem_post()`. К сожалению, в Linux такие семафоры реализованы только для нитей исполнения одного процесса, и поэтому подробно мы на них останавливаться не будем.

Семинар 9. Очереди сообщений в UNIX

Сообщения как средства связи и средства синхронизации процессов. Очереди сообщений в UNIX как составная часть System V IPC. Создание очереди сообщений или доступ к уже существующей. Системный вызов `msgget()`. Реализация примитивов `send` и `receive`. Системные вызовы `msgsnd()` и `msgrcv()`. Удаление очереди сообщений из системы с помощью команды `ipcrm` или системного вызова `msgctl()`. Понятие мультиплексирования. Мультиплексирование сообщений. Модель взаимодействия процессов клиент — сервер. Неравноправность клиента и сервера. Использование очередей сообщений для синхронизации работы процессов.

Ключевые слова: очереди сообщений System V IPC, тип сообщения, выбор сообщений по типам, системные вызовы `msgget()`, `msgsnd()`, `msgrcv()` и `msgctl()`, шаблон сообщения, мультиплексирование сообщений, модель клиент–сервер.

Сообщения как средства связи и средства синхронизации процессов

В материалах предыдущих семинаров были представлены такие средства организации взаимодействия процессов из состава средств System V IPC, как разделяемая память (семинары 6–7) и семафоры (семинар 8). Третьим и последним, наиболее семантически нагруженным средством, входящим в System V IPC, являются *очереди сообщений*. В лекции 6 говорилось о модели сообщений как о способе взаимодействия процессов через линии связи, в котором на передаваемую информацию накладывается определенная структура, так что процесс, принимающий данные, может четко определить, где заканчивается одна порция информации и начинается другая. Такая модель позволяет задействовать одну и ту же линию связи для передачи данных в двух направлениях между несколькими процессами. Мы также рассматривали возможность использования сообщений с встроенными механизмами взаимоисключения и блокировки при чтении из пустого буфера и записи в переполненный буфер для организации синхронизации процессов.

В материалах этого семинара речь пойдет об использовании очередей сообщений System V IPC для обеспечения обеих названных функций.

Очереди сообщений в UNIX как составная часть System V IPC

Так как очереди сообщений входят в состав средств System V IPC, для них верно все, что говорилось ранее об этих средствах в целом, и уже знакомо нам. Очереди сообщений, как и семафоры, и разделяемая память, являются средством связи с непрямой адресацией, требуют инициализации для организации взаимодействия процессов и специальных действий для освобождения системных ресурсов по окончании взаимодействия. Пространством имен очередей сообщений является то же самое множество значений ключа, генерируемых с помощью функции `ftok()` (см. семинары 6–7 раздел «Пространство имен. Адресация в System V IPC. Функция ftok()»). Для выполнения примитивов `send` и `receive`, введенных в лекции 6, соответствующим системным вызовам в качестве параметра передаются IPC-дескрипторы (см. семинары 6–7 раздел «Дескрипторы System V IPC») очередей сообщений, однозначно идентифицирующих их во всей вычислительной системе.

Очереди сообщений располагаются в адресном пространстве ядра операционной системы в виде однонаправленных списков и имеют ограничение по объему информации, хранящейся в каждой очереди. Каждый элемент списка представляет собой отдельное сообщение. Сообщения имеют атрибут, называемый *типом сообщения*. Выборка сообщений из очереди (выполнение примитива `receive`) может осуществляться тремя способами:

1. В порядке FIFO, независимо от типа сообщения.
2. В порядке FIFO для сообщений конкретного типа.
3. Первым выбирается сообщение с минимальным типом, не превышающим некоторого заданного значения, и пришедшее раньше других сообщений с тем же типом.

Реализация примитивов `send` и `receive` обеспечивает скрытое от пользователя взаимоисключение во время помещения сообщения в очередь или его получения из очереди. Также она обеспечивает блокировку процесса при попытке выполнить примитив `receive` над пустой очередью или очередью, в которой отсутствуют сообщения запрошенного типа, или при попытке выполнить примитив `send` для очереди, в которой нет свободного места.

Очереди сообщений, как и другие средства System V IPC, позволяют организовать взаимодействие процессов, не находящихся одновременно в вычислительной системе.

Создание очереди сообщений или доступ к уже существующей. Системный вызов msgget()

Для создания очереди сообщений, ассоциированной с определенным ключом, или доступа по ключу к уже существующей очереди используется системный вызов `msgget()`, являющийся аналогом системных вызовов `shmget()` для разделяемой памяти и `semget()` для массива семафоров, который возвращает значение IPC-дескриптора для этой очереди. При этом существуют те же способы создания и доступа, что и для разделяемой памяти или семафоров (см. семинары 6–7 раздел «Разделяемая память в UNIX. Системные вызовы shmget(), shmat(), shmdt()» и семинар 8 раздел «Создание массива семафоров или доступ к уже существующему. Системный вызов semget()», соответственно).

Системный вызов msgget()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

Описание системного вызова

Системный вызов `msgget` предназначен для выполнения операции доступа к очереди сообщений и, в случае ее успешного завершения, возвращает дескриптор System V IPC для этой очереди (целое неотрицательное число, однозначно характеризующее очередь сообщений внутри вычислительной системы и использующееся в дальнейшем для других операций с ней).

Параметр `key` является ключом System V IPC для очереди сообщений, т. е. фактически ее именем из пространства имен System V IPC. В качестве значения этого параметра может быть использовано значение ключа, полученное с помощью функции `ftok()`, или специальное значение `IPC_PRIVATE`. Использование значения `IPC_PRIVATE` всегда приводит к попытке создания новой очереди сообщений с ключом, который не совпадает со значением ключа ни одной из уже существующих очередей и не может быть получен с помощью функции `ftok()` ни при одной комбинации ее параметров.

Параметр `msgflg` – флаги – играет роль только при создании новой очереди сообщений и определяет права различных пользователей при доступе к очереди, а также необходимость создания новой очереди и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или – « | ») следующих предопределенных значений и восьмеричных прав доступа:

IPC_CREAT – если очереди для указанного ключа не существует, она должна быть создана;

IPC_EXCL – применяется совместно с флагом IPC_CREAT. При совместном их использовании и существовании массива с указанным ключом доступ к очереди не производится и констатируется ошибочная ситуация; при этом переменная errno, описанная в файле <errno.h>, примет значение EEXIST;

0400 – разрешено чтение для пользователя, создавшего очередь;

0200 – разрешена запись для пользователя, создавшего очередь;

0040 – разрешено чтение для группы пользователя, создавшего очередь;

0020 – разрешена запись для группы пользователя, создавшего очередь;

0004 – разрешено чтение для всех остальных пользователей;

0002 – разрешена запись для всех остальных пользователей.

Очередь сообщений имеет ограничение по общему количеству хранимой информации, которое может быть изменено администратором системы. Текущее значение ограничения можно узнать с помощью команды

```
ipcs -l
```

Возвращаемое значение

Системный вызов возвращает значение дескриптора System V IPC для очереди сообщений при нормальном завершении и значение -1 при возникновении ошибки.

Реализация примитивов send и receive. Системные вызовы msgsnd() и msgrcv()

Для выполнения примитива send используется системный вызов msgsnd(), копирующий пользовательское сообщение в очередь сообщений, заданную IPC-дескриптором. При изучении описания этого вызова обратите особое внимание на следующие моменты:

- Тип данных struct msgbuf не является типом данных для пользовательских сообщений, а представляет собой лишь шаблон для создания таких типов. Пользователь сам должен создать структуру для своих сообщений, в которой первым полем должна быть переменная типа long, содержащая положительное значение типа сообщения.
- В качестве третьего параметра – длины сообщения – указывается не вся длина структуры данных, соответствующей сообщению, а только длина полезной информации, т. е. информации, располагающейся в структуре данных после типа сообщения. Это значение может быть и равным 0 в случае, когда вся полезная информация заключается в самом факте прихода сообщения (сообщение используется как сигнальное средство связи).

- В материалах семинаров мы, как правило, будем использовать нулевое значение флага системного вызова, которое приводит к блокировке процесса при отсутствии свободного места в очереди сообщений.

Системный вызов msgsnd()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, struct msgbuf *ptr,
int length, int flag);
```

Описание системного вызова

Системный вызов msgsnd предназначен для помещения сообщения в очередь сообщений, т. е. является реализацией примитива send.

Параметр msqid является дескриптором System V IPC для очереди, в которую отправляется сообщение, т. е. значением, которое вернул системный вызов msgget() при создании очереди или при ее поиске по ключу.

Структура struct msgbuf описана в файле <sys/msg.h> как

```
struct msgbuf {
   long mtype;
   char mtext[1];
};
```

Она представляет собой некоторый шаблон структуры сообщения пользователя. Сообщение пользователя – это структура, первый элемент которой обязательно имеет тип long и содержит тип сообщения, а далее следует информативная часть теоретически произвольной длины (практически в Linux она ограничена размером 4080 байт и может быть еще уменьшена системным администратором), содержащая собственно суть сообщения. Например:

```
struct mymsgbuf {
   long mtype;
   char mtext[1024];
} mybuf;
```

При этом информация вовсе не обязана быть текстовой, например:

```
struct mymsgbuf {
```

```
long mtype;
struct {
int iinfo;
float finfo;
} info;
} mybuf;
```

Тип сообщения должен быть строго положительным числом. Действительная длина полезной части информации (т. е. информации, расположенной в структуре после типа сообщения) должна быть передана системному вызову в качестве параметра length. Этот параметр может быть равен и 0, если вся полезная информация заключается в самом факте наличия сообщения. Системный вызов копирует сообщение, расположенное по адресу, на который указывает параметр ptr, в очередь сообщений, заданную дескриптором msqid.

Параметр flag может принимать два значения: 0 и IPC_NOWAIT. Если значение флага равно 0, и в очереди не хватает места для того, чтобы поместить сообщение, то системный вызов блокируется до тех пор, пока не освободится место. При значении флага IPC_NOWAIT системный вызов в этой ситуации не блокируется, а констатирует возникновение ошибки с установлением значения переменной errno, описанной в файле <errno.h>, равным EAGAIN.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Примитив receive реализуется системным вызовом msgrcv(). При изучении описания этого вызова нужно обратить особое внимание на следующие моменты:

- Тип данных struct msgbuf, как и для вызова msgsnd(), является лишь шаблоном для пользовательского типа данных.
- Способ выбора сообщения (см. раздел «Очереди сообщений в UNIX как составная часть System V IPC» текущего семинара) задается нулевым, положительным или отрицательным значением параметра type. Точное значение типа выбранного сообщения можно определить из соответствующего поля структуры, в которую системный вызов скопирует сообщение.
- Системный вызов возвращает длину только полезной части скопированной информации, т. е. информации, расположенной в структуре после поля типа сообщения.
- Выбранное сообщение удаляется из очереди сообщений.
- В качестве параметра length указывается максимальная длина полезной части информации, которая может быть размещена в структуре, адресованной параметром ptr.

- В материалах семинаров мы будем, как правило, пользоваться нулевым значением флагов для системного вызова, которое приводит к блокировке процесса в случае отсутствия в очереди сообщений с запрошенным типом и к ошибочной ситуации в случае, когда длина информативной части выбранного сообщения превышает длину, специфицированную в параметре `length`.

Системный вызов msgrcv()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv(int msqid, struct msgbuf *ptr,
   int length, long type, int flag);
```

Описание системного вызова

Системный вызов `msgrcv` предназначен для получения сообщения из очереди сообщений, т. е. является реализацией примитива `receive`.

Способ выборки	Значение параметра `type`
В порядке FIFO, независимо от типа сообщения	0
В порядке FIFO для сообщений с типом n	n
Первым выбирается сообщение с минимальным типом, не превышающим значения n, пришедшее ранее всех других сообщений с тем же типом	-n

Параметр `msqid` является дескриптором System V IPC для очереди, из которой должно быть получено сообщение, т. е. значением, которое вернул системный вызов `msgget()` при создании очереди или при ее поиске по ключу.

Параметр `type` определяет способ выборки сообщения из очереди следующим образом. Структура `struct msgbuf` описана в файле <sys/msg.h> как

```
struct msgbuf {
   long mtype;
   char mtext[1];
};
```

Она представляет собой некоторый шаблон структуры сообщения пользователя. Сообщение пользователя – это структура, первый элемент которой обязательно имеет тип long и содержит тип сообщения, а далее следует информативная часть теоретически произвольной длины (практически в Linux она ограничена размером 4080 байт и может быть еще уменьшена системным администратором), содержащая собственно суть сообщения. Например:

```
struct mymsgbuf {
   long mtype;
   char mtext[1024];
} mybuf;
```

При этом информация вовсе не обязана быть текстовой, например:

```
struct mymsgbuf {
   long mtype;
   struct {
      int iinfo;
      float finfo;
   } info;
} mybuf;
```

Параметр length должен содержать максимальную длину полезной части информации (т. е. информации, расположенной в структуре после типа сообщения), которая может быть размещена в сообщении.

В случае удачи системный вызов копирует выбранное сообщение из очереди сообщений по адресу, указанному в параметре ptr, одновременно удаляя его из очереди сообщений.

Параметр flag может принимать значение 0 или быть какой-либо комбинацией флагов IPC_NOWAIT и MSG_NOERROR. Если флаг IPC_NOWAIT не установлен и очередь сообщений пуста или в ней нет сообщений с заказанным типом, то системный вызов блокируется до появления запрошенного сообщения. При установлении флага IPC_NOWAIT системный вызов в этой ситуации не блокируется, а констатирует возникновение ошибки с установлением значения переменной errno, описанной в файле <errno.h>, равным EAGAIN. Если действительная длина полезной части информации в выбранном сообщении превышает значение, указанное в параметре length и флаг MSG_NOERROR не установлен, то выборка сообщения не производится, и фиксируется наличие ошибочной ситуации. Если флаг MSG_NOERROR установлен, то в этом случае ошибки не возникает, а сообщение копируется в сокращенном виде.

Возвращаемое значение

Системный вызов возвращает при нормальном завершении действительную длину полезной части информации (т. е. информации, расположенной в структуре после типа сообщения), скопированной из очереди сообщений, и значение -1 при возникновении ошибки.

Максимально возможная длина информативной части сообщения в операционной системе Linux составляет 4080 байт и может быть уменьшена при генерации системы. Текущее значение максимальной длины можно определить с помощью команды

```
ipcs -l
```

Удаление очереди сообщений из системы с помощью команды ipcrm или системного вызова msgctl()

После завершения процессов, использовавших очередь сообщений, она не удаляется из системы автоматически, а продолжает сохраняться в системе вместе со всеми невостребованными сообщениями до тех пор, пока не будет выполнена специальная команда или специальный системный вызов. Для удаления очереди сообщений можно воспользоваться уже знакомой нам командой `ipcrm`, которая в этом случае примет вид

```
ipcrm msg <IPC идентификатор>
```

Для получения IPC-идентификатора очереди сообщений примените команду `ipcs`. Можно удалить очередь сообщений и с помощью системного вызова `msgctl()`. Этот вызов умеет выполнять и другие операции над очередью сообщений, но в рамках данного курса мы их рассматривать не будем. Если какой-либо процесс находился в состоянии ***ожидание*** при выполнении системного вызова `msgrcv()` или `msgsnd()` для удаляемой очереди, то он будет разблокирован, и системный вызов констатирует наличие ошибочной ситуации.

Системный вызов msgctl()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Описание системного вызова

Системный вызов msgctl предназначен для получения информации об очереди сообщений, изменения ее атрибутов и удаления из системы. Данное описание не является полным описанием системного вызова, а ограничивается рамками текущего курса. Для изучения полного описания обращайтесь к UNIX Manual.

В нашем курсе мы будем пользоваться системным вызовом msgctl только для удаления очереди сообщений из системы. Параметр msqid является дескриптором System V IPC для очереди сообщений, т. е. значением, которое вернул системный вызов msgget() при создании очереди или при ее поиске по ключу.

В качестве параметра cmd в рамках нашего курса мы всегда будем передавать значение IPC_RMID – команду для удаления очереди сообщений с заданным идентификатором. Параметр buf для этой команды не используется, поэтому мы всегда будем подставлять туда значение NULL.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Прогон примера с однонаправленной передачей текстовой информации

Для иллюстрации сказанного рассмотрим две простые программы:

```
/* Программа 09-1a.c для иллюстрации работы с
очередями сообщений */
/* Эта программа получает доступ к очереди
сообщений, отправляет в нее 5 текстовых сообщений
с типом 1 и одно пустое сообщение с типом 255,
которое будет служить для программы 09-1b.c
сигналом прекращения работы. */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdio.h>
#define LAST_MESSAGE 255 /* Тип сообщения для
  прекращения работы программы 09-1b.c */
int main()
{
  int msqid; /* IPC-дескриптор для очереди сообщений */
char pathname[] = "09-1a.c"; /* Имя файла,
```

```
      использующееся для генерации ключа. Файл с таким
      именем должен существовать в текущей директории */
   key_t key; /* IPC ключ */
   int i,len; /* Счетчик цикла и длина
      информативной части сообщения */
   /* Ниже следует пользовательская структура для
      сообщения */
   struct mymsgbuf
   {
      long mtype;
      char mtext[81];
   } mybuf;
   /* Генерируем IPC-ключ из имени файла 09-1a.c в
   текущей директории и номера экземпляра очереди
   сообщений 0. */
   if((key = ftok(pathname,0)) < 0){
      printf("Can\'t generate key\n");
      exit(-1);
   }
   /* Пытаемся получить доступ по ключу к очереди
   сообщений, если она существует, или создать ее,
   с правами доступа read & write для всех
   пользователей */
   if((msqid = msgget(key, 0666 | IPC_CREAT)) < 0){
      printf("Can\'t get msqid\n");
      exit(-1);
   }
   /* Посылаем в цикле пять сообщений с типом 1
   в очередь сообщений, идентифицируемую msqid.*/
   for (i = 1; i <= 5; i++){
      /* Сначала заполняем структуру для нашего
      сообщения и определяем длину информативной части */
      mybuf.mtype = 1;
      strcpy(mybuf.mtext, "This is text message");
      len = strlen(mybuf.mtext)+1;
      /* Отсылаем сообщение. В случае ошибки сообщаем
      об этом и удаляем очередь сообщений из системы. */
      if (msgsnd(msqid, (struct msgbuf *) &mybuf,
         len, 0) < 0){
         printf("Can\'t send message to queue\n");
         msgctl(msqid, IPC_RMID,
            (struct msqid_ds *) NULL);
```

```
            exit(-1);
        }
    }
    /* Отсылаем сообщение, которое заставит получающий
    процесс прекратить работу, с типом LAST_MESSAGE и
    длиной 0 */
    mybuf.mtype = LAST_MESSAGE;
    len = 0;
    if (msgsnd(msqid, (struct msgbuf *) &mybuf,
        len, 0) < 0){
        printf("Can\'t send message to queue\n");
        msgctl(msqid, IPC_RMID,
            (struct msqid_ds *) NULL);
        exit(-1);
    }
    return 0;
}

/* Программа 09-1b.c для иллюстрации работы с
очередями сообщений */
/* Эта программа получает доступ к очереди сообщений и
читает из нее сообщения с любым типом в порядке FIFO
до тех пор, пока не получит сообщение с типом 255,
которое будет служить сигналом прекращения работы. */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdio.h>
#define LAST_MESSAGE 255 /* Тип сообщения для
    прекращения работы */
int main()
{
    int msqid; /* IPC-дескриптор для очереди сообщений */
    char pathname[] = "09-1a.c"; /* Имя файла,
        использующееся для генерации ключа. Файл с таким
        именем должен существовать в текущей директории */
    key_t key; /* IPC ключ */
    int len, maxlen; /* Реальная длина и максимальная
        длина информативной части сообщения */
    /* Ниже следует пользовательская структура для
    сообщения */
```

```
struct mymsgbuf
{
  long mtype;
  char mtext[81];
} mybuf;
/* Генерируем IPC-ключ из имени файла 09-1a.c в
текущей директории и номера экземпляра очереди
сообщений 0 */
if((key = ftok(pathname,0)) < 0){
  printf("Can\'t generate key\n");
  exit(-1);
}
/* Пытаемся получить доступ по ключу к очереди
сообщений, если она существует, или создать ее,
с правами доступа read & write для всех пользо-
вателей */
if((msqid = msgget(key, 0666 | IPC_CREAT)) < 0){
  printf("Can\'t get msqid\n");
  exit(-1);
}
while(1){
  /* В бесконечном цикле принимаем сообщения
  любого типа в порядке FIFO с максимальной длиной
  информативной части 81 символ до тех пор, пока
  не поступит сообщение с типом LAST_MESSAGE*/
  maxlen = 81;
  if(( len = msgrcv(msqid,
    (struct msgbuf *) &mybuf, maxlen, 0, 0) < 0){
    printf("Can\'t receive message from queue\n");
    exit(-1);
  }
  /* Если принятое сообщение имеет тип LAST_MESSAGE,
  прекращаем работу и удаляем очередь сообщений из
  системы. В противном случае печатаем текст
  принятого сообщения. */
  if (mybuf.mtype == LAST_MESSAGE){
    msgctl(msqid, IPC_RMID,
    (struct msqid_ds *) NULL);
  exit(0);
  }
  printf("message type = %ld, info = %s\n",
  mybuf.mtype, mybuf.mtext);
```

```
    }
    return 0; /* Исключительно для отсутствия
      warning'ов при компиляции. */
}
```

Первая из этих программ посылает пять текстовых сообщений с типом 1 и одно сообщение нулевой длины с типом 255 второй программе. Вторая программа в цикле принимает сообщения любого типа в порядке FIFO и печатает их содержимое до тех пор, пока не получит сообщение с типом 255. Сообщение с типом 255 служит для нее сигналом к завершению работы и ликвидации очереди сообщений. Если перед запуском любой из программ очередь сообщений еще отсутствовала в системе, то программа создаст ее.

Обратите внимание на использование сообщения с типом 255 в качестве сигнала прекращения работы второго процесса. Это сообщение имеет нулевую длину, так как его информативность исчерпывается самим фактом наличия сообщения.

Наберите программы, сохраните под именами 09-1a.c и 09-1b.c соответственно, откомпилируйте и проверьте правильность их поведения.

Модификация предыдущего примера для передачи числовой информации

В описании системных вызовов msgsnd() и msgrcv() говорится о том, что передаваемая информации не обязательно должна представлять собой текст.

Мы можем воспользоваться очередями сообщений для передачи данных любого вида. При передаче разнородной информации целесообразно информативную часть объединять внутри сообщения в отдельную структуру:

```
struct mymsgbuf {
    long mtype;
    struct {
        short sinfo;
        float finfo;
    } info;
} mybuf;
```

для правильного вычисления длины информативной части. В некоторых вычислительных системах числовые данные размещаются в памяти с выравниванием на определенные адреса (например, на адреса, кратные 4).

Поэтому реальный размер памяти, необходимой для размещения нескольких числовых данных, может оказаться больше суммы длин этих данных, т. е. в нашем случае

```
sizeof(info)>=sizeof(short)+sizeof(float)
```

Для полной передачи информативной части сообщения в качестве длины нужно указывать не сумму длин полей, а полную длину структуры. Модифицируйте предыдущие программы 09-1a.c и 09-1b.c из раздела «Прогон примера с однонаправленной передачей текстовой информации» для передачи нетекстовых сообщений.

Написание, компиляция и прогон программ для осуществления двусторонней связи через одну очередь сообщений

Наличие у сообщений типов позволяет организовать двустороннюю связь между процессами через одну и ту же очередь сообщений. Процесс 1 может посылать процессу 2 сообщения с типом 1, а получать от него сообщения с типом 2. При этом для выборки сообщений в обоих процессах следует пользоваться вторым способом выбора (см. раздел «Очереди сообщений в UNIX как составная часть System V IPC»). Напишите, откомпилируйте и прогоните программы, осуществляющие двустороннюю связь через одну очередь сообщений.

Понятие мультиплексирования. Мультиплексирование сообщений. Модель взаимодействия процессов клиент — сервер. Неравноправность клиента и сервера

Используя технику из предыдущего примера, мы можем организовать получение сообщений одним процессом от множества других процессов через одну очередь сообщений и отправку им ответов через ту же очередь сообщений, т. е. осуществить *мультиплексирование* сообщений. Вообще под мультиплексированием информации понимают возможность одновременного обмена информацией с несколькими партнерами. Метод мультиплексирования широко применяется в модели взаимодействия процессов клиент–сервер. В этой модели один из процессов является сервером. Сервер получает запросы от других процессов – клиентов – на выполнение некоторых действий и отправляет им результаты

обработки запросов. Чаще всего модель клиент–сервер используется при разработке сетевых приложений, с которыми мы столкнемся в материалах завершающих семинаров курса. Она изначально предполагает, что взаимодействующие процессы неравноправны:

- Сервер, как правило, работает постоянно, на всем протяжении жизни приложения, а клиенты могут работать эпизодически.
- Сервер ждет запроса от клиентов, инициатором же взаимодействия является клиент.
- Как правило, клиент обращается к одному серверу за раз, в то время как к серверу могут одновременно поступать запросы от нескольких клиентов.
- Клиент должен знать, как обратиться к серверу (например, какого типа сообщения он воспринимает) перед началом организации запроса к серверу, в то время как сервер может получить недостающую информацию о клиенте из пришедшего запроса.

Рассмотрим следующую схему мультиплексирования сообщений через одну очередь сообщений для модели клиент–сервер. Пусть сервер получает из очереди сообщений только сообщения с типом `1`. В состав сообщений с типом `1`, посылаемых серверу, процессы-клиенты включают значения своих идентификаторов процесса. Приняв сообщение с типом `1`, сервер анализирует его содержание, выявляет идентификатор процесса, пославшего запрос, и отвечает клиенту, посылая сообщение с типом, равным идентификатору запрашивавшего процесса. Процесс-клиент после отправления запроса ожидает ответа в виде сообщения с типом, равным своему идентификатору. Поскольку идентификаторы процессов в системе различны, и ни один пользовательский процесс не может иметь `PID` равный `1`, все сообщения могут быть прочитаны только теми процессами, которым они адресованы. Если обработка запроса занимает продолжительное время, сервер может организовывать параллельную обработку запросов, порождая для каждого запроса новый процесс-ребенок или новую нить исполнения.

Написание, компиляция и прогон программ клиента и сервера

Напишите, откомпилируйте и прогоните программы сервера и клиентов для предложенной схемы мультиплексирования сообщений.

Использование очередей сообщений для синхронизации работы процессов

В лекции 6 была доказана эквивалентность очередей сообщений и семафоров в системах, где процессы могут использовать разделяемую память. В частности, было показано, как реализовать семафоры с помощью очередей сообщений. Для этого вводился специальный синхронизирующий процесс-сервер, обслуживающий переменные-счетчики для каждого семафора. Процессы-клиенты для выполнения операции над семафором посылали процессу-серверу запросы на выполнение операции и ожидали ответа для продолжения работы. Теперь мы знаем, как это можно сделать в операционной системе UNIX и как, следовательно, можно использовать очереди сообщений для организации взаимоисключений и взаимной синхронизации процессов.

Задача повышенной сложности: реализуйте семафоры через очереди сообщений.

Семинары 10–11. Организация файловой системы в UNIX. Работа с файлами и директориями. Понятие о memory mapped файлах

Разделы носителя информации (partitions) в UNIX. Логическая структура файловой системы и типы файлов в UNIX. Организация файла на диске в UNIX на примере файловой системы s5fs. Понятие индексного узла (inode). Организация директорий (каталогов) в UNIX. Понятие суперблока. Операции над файлами и директориями. Системные вызовы и команды для выполнения операций над файлами и директориями. Системный вызов `open()`. Системный вызов `close()`. Операция создания файла. Системный вызов `creat()`. Операция чтения атрибутов файла. Системные вызовы `stat()`, `fstat()` и `lstat()`. Операции изменения атрибутов файла. Операции чтения из файла и записи в файл. Операция изменения указателя текущей позиции. Системный вызов `lseek()`. Операция добавления информации в файл. Флаг `O_APPEND`. Операции создания связей. Команда `ln`, системные вызовы `link()` и `symlink()`. Операция удаления связей и файлов. Системный вызов `unlink()`. Специальные функции для работы с содержимым директорий. Понятие о файлах, отображаемых в память (memory mapped файлах). Системные вызовы `mmap()`, `munmap()`.

Ключевые слова: разделы (partitions) или логические диски, регулярные файлы, директории, файлы типа FIFO, файлы устройств, файлы типа «связь», файлы типа «сокет», граф файловой системы, корневая директория, полное имя файла, файловая система `s5fs`, заголовок раздела, индексный узел (inode), массив индексных узлов, атрибуты файла, суперблок, операции над файлами, указатель текущей позиции, таблица открытых файлов процесса, системная таблица открытых файлов, таблица индексных узлов открытых файлов, жесткая связь, мягкая или символическая связь, команды `chmod`, `chgrp`, `chown`, `cp`, `rm`, `ls`, `mv`, `ln`, системные вызовы `open()`, `close()`, `read()`, `write()`, `create()`, `stat()`, `lstat()`, `fstat()`, `lseek()`, `link()`, `symlink()`, `unlink()`, `ftruncate()`, операции над директориями, функции `opendir()`, `readdir()`, `rewinddir()`, `closedir()`, файлы, отображаемые в память (memory mapped файлы), системные вызовы `mmap()`, `munmap()`.

Введение

В материалах нескольких предыдущих семинаров (семинары 1–2, семинар 5) уже затрагивались вопросы работы с файлами в UNIX. Но только теперь, пояснив в лекции понятие файловой системы, мы можем рассмотреть файловую систему UNIX в целом. Наш обзор, правда, ограничится общими вопросами, связанными с организацией файловой системы, и системными вызовами, которые с наибольшей вероятностью могут пригодиться в дальнейшем. Это связано как с ограниченностью времени, которое отводится на работу с файловыми системами в нашем курсе, так и с преимущественно практическим направлением наших занятий.

Разделы носителя информации (partitions) в UNIX

Физические носители информации – магнитные или оптические диски, ленты и т. д., использующиеся как физическая основа для хранения файлов, в операционных системах принято логически делить на *разделы* (*partitions*) или *логические диски*. Причем слово «делить» не следует понимать буквально, в некоторых системах несколько физических дисков могут быть объединены в один раздел. Об этом подробнее рассказывается в лекции 12 в разделе «Общая структура файловой системы».

В операционной системе UNIX физический носитель информации обычно представляет собой один или несколько разделов. В большинстве случаев разбиение на разделы производится линейно, хотя некоторые варианты UNIX могут допускать некое подобие древовидного разбиения (Solaris). Количество разделов и их размеры определяются при форматировании диска. Поскольку форматирование диска относится к области администрирования операционных систем, оно в нашем курсе рассматриваться не будет.

Наличие нескольких разделов на диске может определяться требованиями операционной системы или пожеланиями пользователя. Допустим, пользователь хочет разместить на одном жестком диске несколько операционных систем с возможностью попеременной работы в них, тогда он размещает каждую операционную систему в своем разделе. Или другая ситуация: необходимость работы с несколькими видами файловых систем. Под каждый тип файловой системы выделяется отдельный логический диск. Третий вариант – это разбиение диска на разделы для размещения в разных разделах различных категорий файлов. Скажем, в одном разделе помещаются все системные файлы, а в другом разделе – все пользовательские файлы. Примером операционной системы, внутренние требования которой приводят к появлению нескольких разделов на диске, могут служить ранние версии MS-DOS, для которых максимальный размер логического диска не превышал 32 Мбайт.

Для простоты далее в этих семинарах будем полагать, что у нас имеется только один раздел и, следовательно, одна файловая система. Вопросы взаимного сосуществования нескольких файловых систем в рамках одной операционной системы мы затронем в семинарах 13–14 перед обсуждением реализации подсистемы ввода-вывода.

Логическая структура файловой системы и типы файлов в UNIX

Мы не будем давать здесь определение файла, полагая, что интуитивное представление о файлах у вас имеется, а в лекции 11 (раздел «Введение») было введено понятие о файлах как об именованных абстрактных объектах, обладающих определенными свойствами. При этом в пространстве имен файлов одному файлу могут соответствовать несколько имен.

В материалах семинаров 1–2 упрощенно говорилось о том, что файлы могут объединяться в директории, и что файлы и директории организованы в древовидную структуру. На нынешнем уровне знаний мы можем сформулировать это более аккуратно. В операционной системе UNIX существуют файлы нескольких типов, а именно:

- обычные или регулярные файлы;
- директории или каталоги;
- файлы типа FIFO или именованные pip'ы;
- специальные файлы устройств;
- сокеты (sockets);
- специальные файлы связи (link).

Что такое регулярные файлы и директории, вам должно быть хорошо известно из личного опыта и из лекций (лекция 11). О способах их отображения в дисковое пространство речь пойдет чуть позже. Файлы типа FIFO были представлены в семинаре 5, когда рассматривалась работа с именованными pip'ами (раздел «Понятие FIFO. Использование системного вызова mknod() для создания FIFO. Функция mkfifo()»). Файлы типа «связь» мы представим в этом семинаре, когда будем обсуждать операции над файлами (раздел «Операции над файлами и директориями») и соответствующие им системные вызовы (раздел «Системные вызовы и команды для выполнения операций над файлами и директориями»). О специальных файлах устройств будет рассказано в материалах семинаров 12–13, посвященных реализации в UNIX подсистемы ввода-вывода и передаче информации с помощью сигналов. Файлы типа «сокет» будут введены в семинарах 14–15, когда мы будем рассматривать вопросы сетевого программирования в UNIX.

Файлы всех перечисленных типов логически объединены в ациклический граф с однонаправленными ребрами, получающийся из дерева в

результате сращивания нескольких терминальных узлов дерева или нескольких его нетерминальных узлов таким образом, чтобы полученный граф не содержал циклов. В нетерминальных узлах такого ациклического графа (т. е. в узлах, из которых выходят ребра) могут располагаться только файлы типов «директория» и «связь». Причем из узла, в котором располагается файл типа «связь», может выходить только одно ребро. В терминальных узлах этого ациклического графа (т. е. в узлах, из которых не выходит ребер) могут располагаться файлы любых типов (см. рис. 10-11.1), хотя присутствие в терминальном узле файла типа «связь» обычно говорит о некотором нарушении целостности файловой системы.

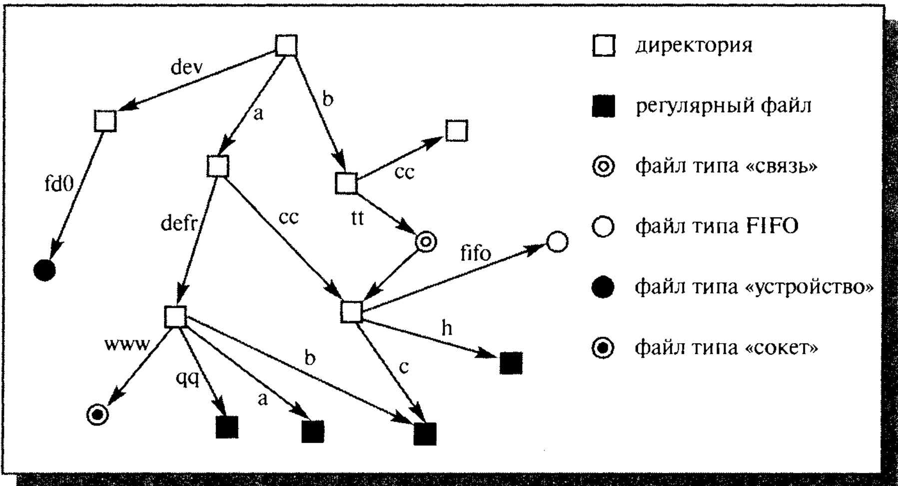


Рис. 10-11.1. Пример графа файловой системы

В отличие от древовидной структуры набора файлов, где имена файлов связывались с узлами дерева, в таком ациклическом графе имя файла связывается не с узлом, соответствующим файлу, а с входящим в него ребром. Ребра, выходящие из узлов, соответствующих файлам типа «связь», являются неименованными. Надо отметить, что практически во всех существующих реализациях UNIX-подобных систем в узел графа, соответствующий файлу типа «директория», не может входить более одного именованного ребра, хотя стандарт на операционную систему UNIX и не запрещает этого. Правила построения имен ребер (файлов) рассматривались в семинарах 1–2. В качестве полного имени файла может использоваться любое имя, получающееся при прохождении по ребрам от корневого узла графа (т. е. узла, в который не входит ни одно ребро) до узла, соответствующего этому файлу, по любому пути с помощью следующего алгоритма:

1. Если интересующему нас файлу соответствует корневой узел, то файл имеет имя «/».
2. Берем первое именованное ребро в пути и записываем его имя, которому предваряем символ «/».
3. Для каждого очередного именованного ребра в пути приписываем к уже получившейся строке справа символ «/» и имя соответствующего ребра.

Полное имя является уникальным для всей файловой системы и однозначно определяет соответствующий ему файл.

Организация файла на диске в UNIX на примере файловой системы s5fs. Понятие индексного узла (inode)

Рассмотрим, как организуется на физическом носителе любой файл в UNIX на примере простой файловой системы, впервые появившейся в вариантах операционной системы System V и носящей поэтому название s5fs (system V file system).

Все дисковое пространство раздела в файловой системе s5fs логически разделяется на две части: *заголовок раздела* и *логические блоки данных*. Заголовок раздела содержит служебную информацию, необходимую для работы файловой системы, и обычно располагается в самом начале раздела. Логические блоки хранят собственно содержательную информацию файлов и часть информации о размещении файлов на диске (т. е. какие логические блоки и в каком порядке содержат информацию, записанную в файл).

Для размещения любого файла на диске используется метод индексных узлов (inode – от index node), о котором подробно рассказывается в лекции 12 (раздел «Методы выделения дискового пространства»), и на котором здесь мы останавливаться не будем. Индексный узел содержит атрибуты файла и оставшуюся часть информации о его размещении на диске. Необходимо, однако, отметить, что такие типы файлов, как «связь», «сокет», «устройство», «FIFO» не занимают на диске никакого иного места, кроме индексного узла (им не выделяется логических блоков). Все необходимое для работы с этими типами файлов содержится в их атрибутах.

Перечислим часть атрибутов файлов, хранящихся в индексном узле и свойственных большинству типов файлов. К таким атрибутам относятся:

- Тип файла и права различных категорий пользователей для доступа к нему.
- Идентификаторы владельца-пользователя и владельца-группы.

- Размер файла в байтах (только для регулярных файлов, директорий и файлов типа «связь»).
- Время последнего доступа к файлу.
- Время последней модификации файла.
- Время последней модификации самого индексного узла.

Существует еще один атрибут, о котором мы поговорим в этих семинарах позже, когда будем рассматривать операцию связывания файлов в разделе «Системные вызовы и команды для выполнения операций над файлами и директориями». Количество индексных узлов в разделе является постоянной величиной, определяемой на этапе генерации файловой системы. Все индексные узлы системы организованы в виде массива, хранящегося в заголовке раздела. Каждому файлу соответствует только один элемент этого массива и, наоборот, каждому непустому элементу этого массива соответствует только один файл. Таким образом, каждый файл на диске может быть однозначно идентифицирован номером своего индексного узла (его индексом в массиве).

На языке представления логической организации файловой системы в виде графа это означает, что каждому узлу графа соответствует только один номер индексного узла, и никакие два узла графа не могут иметь одинаковые номера.

Надо отметить, что свойством уникальности номеров индексных узлов, идентифицирующих файлы, мы уже неявно пользовались при работе с именоваными pip'ами (семинар 5, раздел «Понятие FIFO. Использование системного вызова mknod() для создания FIFO. Функция mkfifo()») и средствами System V IPC (семинары 6–7, раздел «Понятие о System V IPC»). Для именованного pip'a именно номер индексного узла, соответствующего файлу с типом FIFO, является той самой точкой привязки, пользуясь которой, неродственные процессы могут получить данные о расположении pip'a в адресном пространстве ядра и его состоянии и связаться друг с другом. Для средств System V IPC при генерации IPC-ключа с помощью функции `ftok()` в действительности используется не имя заданного файла, а номер соответствующего ему индексного дескриптора, который по определенному алгоритму объединяется с номером экземпляра средства связи.

Организация директорий (каталогов) в UNIX

Содержимое регулярных файлов (информация, находящаяся в них, и способ ее организации) всецело определяется программистом, создающим файл. В отличие от регулярных, остальные типы файлов, содержащих данные, т. е. директории и связи, имеют жестко заданную структуру и содержание, определяемые типом используемой файловой системы.

Основным содержимым файлов типа «директория», если говорить на пользовательском языке, являются имена файлов, лежащих непосредственно в этих директориях, и соответствующие им номера индексных узлов. В терминах представления в виде графа содержимое директорий представляет собой имена ребер, выходящих из узлов, соответствующих директориям, вместе с индексными номерами узлов, к которым они ведут.

В файловой системе s5fs пространство имен файлов (ребер) содержит имена длиной не более 14 символов, а максимальное количество inode в одном разделе файловой системы не может превышать значения 65535. Эти ограничения не позволяют давать файлам осмысленные имена и приводят к необходимости разбиения больших жестких дисков на несколько разделов. Зато они помогают упростить структуру хранения информации в директории. Все содержимое директории представляет собой таблицу, в которой каждый элемент имеет фиксированный размер в 16 байт. Из них 14 байт отводится под имя соответствующего файла (ребра), а 2 байта – под номер его индексного узла. При этом первый элемент таблицы дополнительно содержит ссылку на саму данную директорию под именем «.», а второй элемент таблицы – ссылку на родительский каталог (если он существует), т. е. на узел графа, из которого выходит единственное именованное ребро, ведущее к текущему узлу, под именем «..».

В более современной файловой системе FFS (Fast File System) размерность пространства имен файлов (ребер) увеличена до 255 символов. Это позволило использовать практически любые мыслимые имена для файлов (вряд ли найдется программист, которому будет не лень набирать для имени более 255 символов), но пришлось изменить структуру каталога (чтобы уменьшить его размеры и не хранить пустые байты). В системе FFS каталог представляет собой таблицу из записей переменной длины. В структуру каждой записи входят: номер индексного узла, длина этой записи, длина имени файла и собственно его имя. Две первых записи в каталоге, как и в s5fs, по-прежнему адресуют саму данную директорию и ее родительский каталог.

Понятие суперблока

Мы уже коснулись содержимого заголовка раздела, когда говорили о массиве индексных узлов файловой системы. Оставшуюся часть заголовка в s5fs принято называть суперблоком. Суперблок хранит информацию, необходимую для правильного функционирования файловой системы в целом. В нем содержатся, в частности, следующие данные:

- Тип файловой системы.
- Флаги состояния файловой системы.
- Размер логического блока в байтах (обычно кратен 512 байтам).

- Размер файловой системы в логических блоках (включая сам суперблок и массив inode).
- Размер массива индексных узлов (т. е. сколько файлов может быть размещено в файловой системе).
- Число свободных индексных узлов (сколько файлов еще можно создать).
- Число свободных блоков для размещения данных.
- Часть списка свободных индексных узлов.
- Часть списка свободных блоков для размещения данных.

В некоторых модификациях файловой системы s5fs последние два списка выносятся за пределы суперблока, но остаются в заголовке раздела. При первом же обращении к файловой системе суперблок обычно целиком считывается в адресное пространство ядра для ускорения последующих обращений. Поскольку количество логических блоков и индексных узлов в файловой системе может быть весьма большим, нецелесообразно хранить списки свободных блоков и узлов в суперблоке полностью. При работе с индексными узлами часть списка свободных узлов, находящаяся в суперблоке, постепенно убывает. Когда список почти исчерпан, операционная система сканирует массив индексных узлов и заново заполняет список. Часть списка свободных логических блоков, лежащая в суперблоке, содержит ссылку на продолжение списка, расположенное где-либо в блоках данных. Когда эта часть оказывается использованной, операционная система загружает на освободившееся место продолжение списка, а блок, применявшийся для его хранения, переводится в разряд свободных.

Операции над файлами и директориями

Хотя с точки зрения пользователя рассмотрение операций над файлами и директориями представляется достаточно простым и сводится к перечислению ряда системных вызовов и команд операционной системы, попытка систематического подхода к набору операций вызывает определенные затруднения. Далее речь пойдет в основном о регулярных файлах и файлах типа «директория».

В лекции (лекция 11, раздел «Организация файлов и доступ к ним») речь шла о том, что существует два основных вида файлов, различающихся по методу доступа: файлы последовательного доступа и файлы прямого доступа. Если рассматривать файлы прямого и последовательного доступа как абстрактные типы данных, то они представляются как нечто, содержащее информацию, над которой можно совершать следующие операции:

- Для последовательного доступа: чтение очередной порции данных (`read`), запись очередной порции данных (`write`) и позиционирование на начале файла (`rewind`).

- Для прямого доступа: чтение очередной порции данных (read), запись очередной порции данных (write) и позиционирование на требуемой части данных (seek).

Работа с объектами этих абстрактных типов подразумевает наличие еще двух необходимых операций: создание нового объекта (new) и уничтожение существующего объекта (free).

Расширение математической модели файла за счет добавления к хранимой информации атрибутов, присущих файлу (права доступа, учетные данные), влечет за собой появление еще двух операций: прочитать атрибуты (get attribute) и установить их значения (set attribute).

Наделение файлов какой-либо внутренней структурой (как у файла типа «директория») или наложение на набор файлов внешней логической структуры (объединение в ациклический направленный граф) приводит к появлению других наборов операций, составляющих интерфейс работы с файлами, которые, тем не менее, будут являться комбинациями перечисленных выше базовых операций.

Для директории, например, такой набор операций, определяемый ее внутренним строением, может выглядеть так: операции new, free, set attribute и get attribute остаются без изменений, а операции read, write и rewind (seek) заменяются более высокоуровневыми:

- прочитать запись, соответствующую имени файла, – get record;
- добавить новую запись – add record;
- удалить запись, соответствующую имени файла, – delete record.

Неполный набор операций над файлами, связанный с их логическим объединением в структуру директорий, будет выглядеть следующим образом:

- Операции для работы с атрибутами файлов – get attribute, set attribute.
- Операции для работы с содержимым файлов – read, write, rewind(seek) для регулярных файлов и get record, add record, delete record для директорий.
- Операция создания регулярного файла в некоторой директории (создание нового узла графа и добавление в граф нового именованного ребра, ведущего в этот узел из некоторого узла, соответствующего директории) – create. Эту операцию можно рассматривать как суперпозицию двух операций: базовой операции new для регулярного файла и add record для соответствующей директории.
- Операция создания поддиректории в некоторой директории – make directory. Эта операция отличается от предыдущей операции create занесением в файл новой директории информации о файлах с именами «.» и «..», т. е. по сути дела она есть суперпозиция операции create и двух операций add record.

- Операция создания файла типа «связь» – symbolic link.
- Операция создания файла типа «FIFO» – make FIFO.
- Операция добавления к графу нового именованного ребра, ведущего от узла, соответствующего директории, к узлу, соответствующему любому другому типу файла, – link. Это просто add record с некоторыми ограничениями.
- Операция удаления файла, не являющегося директорией или «связью» (удаление именованного ребра из графа, ведущего к терминальной вершине с одновременным удалением этой вершины, если к ней не ведут другие именованные ребра), – unlink.
- Операция удаления файла типа «связь» (удаление именованного ребра, ведущего к узлу, соответствующему файлу типа «связь», с одновременным удалением этого узла и выходящего из него неименованного ребра, если к этому узлу не ведут другие именованные ребра), – unlink link.
- Операция рекурсивного удаления директории со всеми входящими в нее файлами и поддиректориями – remove directory.
- Операция переименования файла (ребра графа) – rename.
- Операция перемещения файла из одной директории в другую (перемещается точка выхода именованного ребра, которое ведет к узлу, соответствующему данному файлу) – move.

Возможны и другие подобные операции.

Способ реализации файловой системы в реальной операционной системе также может добавлять новые операции. Если часть информации файловой системы или отдельного файла кэшируется в адресном пространстве ядра, то появляются операции синхронизации данных в кэше и на диске для всей системы в целом (sync) и для отдельного файла (sync file).

Естественно, что все перечисленные операции могут быть выполнены процессом только при наличии у него определенных полномочий (прав доступа и т. д.). Для выполнения операций над файлами и директориями операционная система предоставляет процессам интерфейс в виде системных вызовов, библиотечных функций и команд операционной системы. Часть этих системных вызовов, функций и команд мы рассмотрим в следующих разделах.

Системные вызовы и команды для выполнения операций над файлами и директориями

В материалах предыдущих семинаров уже говорилось о некоторых командах и системных вызовах, позволяющих выполнять операции над файлами в операционной системе UNIX.

В семинарах 1–2 рассматривался ряд команд, позволяющих изменять атрибуты файла – chmod, chown, chgrp, команду копирования файлов и директорий – cp, команду удаления файлов и директорий – rm, команду переименования и перемещения файлов и директорий – mv, команду просмотра содержимого директорий – ls.

В материалах семинара 5, посвященного потокам ввода-вывода, рассказывалось о хранении информации о файлах внутри адресного пространства процесса с помощью таблицы открытых файлов, о понятии файлового дескриптора, о необходимости введения операций открытия и закрытия файлов (системные вызовы open() и close()) и об операциях чтения и записи (системные вызовы read() и write()). Мы обещали вернуться к более подробному рассмотрению затронутых вопросов в текущих семинарах. Пора выполнять обещанное. Далее в этом разделе, если не будет оговорено особо, под словом «файл» будет подразумеваться регулярный файл.

Вся информация об атрибутах файла и его расположении на физическом носителе содержится в соответствующем файлу индексном узле и, возможно, в нескольких связанных с ним логических блоках. Для того чтобы при каждой операции над файлом не считывать эту информацию с физического носителя заново, представляется логичным, считав информацию один раз при первом обращении к файлу, хранить ее в адресном пространстве процесса или в части адресного пространства ядра, характеризующей данный процесс. Именно поэтому в лекции 2 данные о файлах, используемых процессом, были отнесены к составу системного контекста процесса, содержащегося в его PCB.

С точки зрения пользовательского процесса каждый файл представляет собой линейный набор байт, снабженный *указателем текущей позиции* процесса в этом наборе. Все операции чтения из файла и записи в файл производятся в этом наборе с того места, на которое показывает указатель текущей позиции. По завершении операции чтения или записи указатель текущей позиции помещается после конца прочитанного или записанного участка файла. Значение этого указателя является динамической характеристикой файла для использующего его процесса и также должно храниться в PCB.

На самом деле организация информации, описывающей открытые файлы в адресном пространстве ядра операционной системы UNIX, является более сложной.

Некоторые файлы могут использоваться одновременно несколькими процессами независимо друг от друга или совместно. Для того чтобы не хранить дублирующуюся информацию об атрибутах файлов и их расположении на внешнем носителе для каждого процесса отдельно, такие данные обычно размещаются в адресном пространстве ядра операцион-

ной системы в единственном экземпляре, а доступ к ним процессы получают только при выполнении соответствующих системных вызовов для операций над файлами.

Независимое использование одного и того же файла несколькими процессами в операционной системе UNIX предполагает возможность для каждого процесса совершать операции чтения и записи в файл по своему усмотрению. При этом для корректной работы с информацией необходимо организовывать взаимоисключения для операций ввода-вывода. Совместное использование одного и того же файла в операционной системе UNIX возможно для близко родственных процессов, т. е. процессов, один из которых является потомком другого или которые имеют общего родителя. При совместном использовании файла процессы разделяют некоторые данные, необходимые для работы с файлом, в частности, указатель текущей позиции. Операции чтения или записи, выполненные в одном процессе, изменяют значение указателя текущей позиции во всех близко родственных процессах, одновременно использующих этот файл.

Как мы видим, вся информация о файле, необходимая процессу для работы с ним, может быть разбита на три части:

- данные, специфичные для этого процесса;
- данные, общие для близко родственных процессов, совместно использующих файл, например, указатель текущей позиции;
- данные, являющиеся общими для всех процессов, использующих файл, – атрибуты и расположение файла.

Естественно, что для хранения этой информации применяются три различные связанные структуры данных, лежащие, как правило, в адресном пространстве ядра операционной системы, – *таблица открытых файлов процесса*, *системная таблица открытых файлов* и *таблица индексных узлов открытых файлов*. Для доступа к этой информации в управляющем блоке процесса заводится таблица открытых файлов, каждый непустой элемент которой содержит ссылку на соответствующий элемент системной таблицы открытых файлов, содержащей данные, необходимые для совместного использования файла близко родственными процессами. Из системной таблицы открытых файлов мы, в свою очередь, можем по ссылке добраться до общих данных о файле, содержащихся в таблице индексных узлов открытых файлов (см. рис. 10-11.2). Только таблица открытых файлов процесса входит в состав его PCB и, соответственно, наследуется при рождении нового процесса. Индекс элемента в этой таблице (небольшое целое неотрицательное число) или файловый дескриптор является той величиной, характеризующей файл, которой может оперировать процесс при работе на уровне пользователя. В эту же таблицу открытых файлов помещаются и ссылки на данные, описывающие другие потоки ввода-вывода, такие как pipe и FIFO (об этом уже упоминалось в

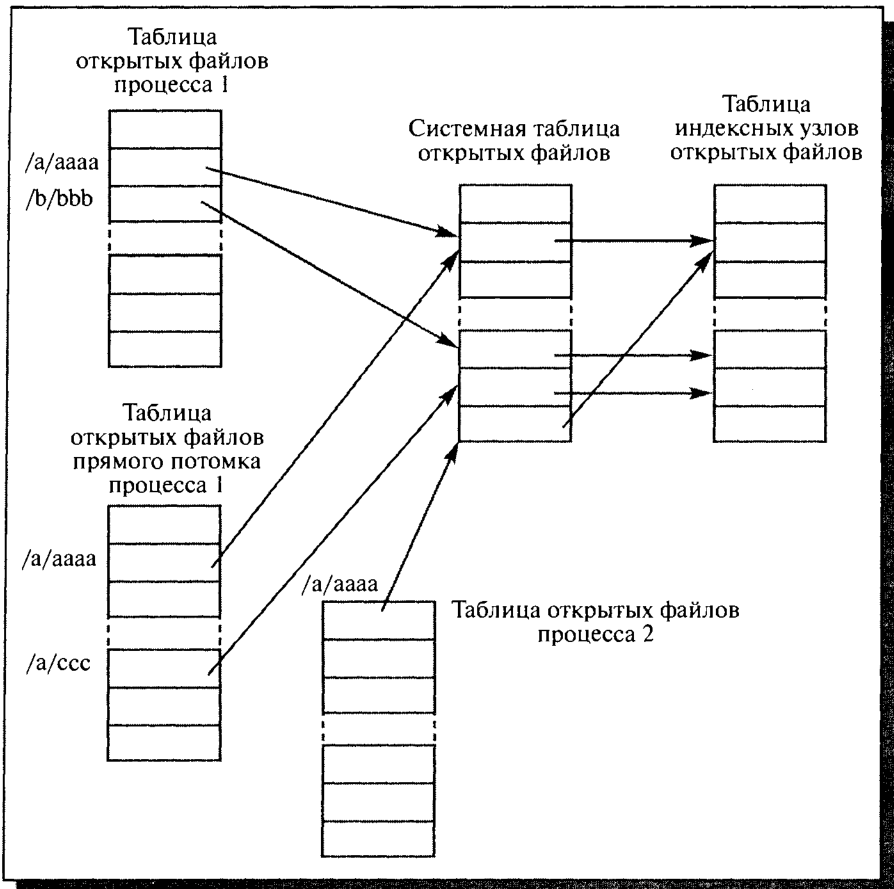


Рис. 10-11.2. Взаимосвязи между таблицами, содержащими данные об открытых файлах в системе

семинаре 5). Как мы увидим позже (в материалах семинаров 15–16, посвященных сетевому программированию), эта же таблица будет использоваться и для размещения ссылок на структуры данных, необходимых для передачи информации от процесса к процессу по сети.

Системный вызов open(). Для выполнения большинства операций над файлами через системные вызовы пользовательский процесс обычно должен указать в качестве одного из параметров системного вызова дескриптор файла, над которым нужно совершить операцию. Поэтому, прежде чем совершать операции, мы должны поместить информацию о файле в наши таблицы файлов и определить соответствующий файловый дескриптор. Для этого, как уже говорилось в семинаре 5, применяется проце-

дура открытия файла, осуществляемая системным вызовом open(). При открытии файла операционная система проверяет, соответствуют ли права, которые запросил процесс для операций над файлом, правам доступа, установленным для этого файла. В случае соответствия она помещает необходимую информацию в системную таблицу файлов и, если этот файл не был ранее открыт другим процессом, в таблицу индексных дескрипторов открытых файлов. Далее операционная система находит пустой элемент в таблице открытых файлов процесса, устанавливает необходимую связь между всеми тремя таблицами и возвращает на пользовательский уровень дескриптор этого файла.

По сути дела, с помощью операции открытия файла операционная система осуществляет отображение из пространства имен файлов в дисковое пространство файловой системы, подготавливая почву для выполнения других операций.

Системный вызов close(). Обратным системным вызовом по отношению к системному вызову open() является системный вызов close(), с которым мы уже познакомились. После завершения работы с файлом процесс освобождает выделенные ресурсы операционной системы и, возможно, синхронизирует информацию о файле, содержащуюся в таблице индексных узлов открытых файлов, с информацией на диске, используя этот системный вызов. Надо отметить, что место в таблице индексных узлов открытых файлов не освобождается по системному вызову close() до тех пор, пока в системе существует хотя бы один процесс, использующий этот файл. Для обеспечения такого поведения в ней для каждого индексного узла заводится счетчик числа открытий, увеличивающийся на 1 при каждом системном вызове open() для данного файла и уменьшающийся на 1 при каждом его закрытии. Очищение элемента таблицы индексных узлов открытых файлов с окончательной синхронизацией данных в памяти и на диске происходит только в том случае, если при очередном закрытии файла этот счетчик становится равным 0.

Поведение таблицы открытых файлов процесса и связанных с ней таблиц при системных вызовах exit(), exec() и fork() рассматривалось в материалах семинара 5.

Операция создания файла. Системный вызов creat(). При обсуждении системного вызова open() подробно рассказывалось о его использовании для создания нового файла. Для этих же целей можно использовать системный вызов creat(), являющийся, по существу, урезанным вариантом вызова open() (о значении флага O_TRUNC для системного вызова open() будет сказано чуть ниже).

Системный вызов creat()

Прототип системного вызова

```
#include <fcntl.h>

int creat(char *path, int mode);
```

Описание системного вызова

Системный вызов `creat` эквивалентен системному вызову `open()` с параметром `flags`, установленным в значение `O_CREAT | O_WRONLY | O_TRUNC`.

Параметр `path` является указателем на строку, содержащую полное или относительное имя файла.

Если файла с указанным именем не существовало к моменту системного вызова, он будет создан и открыт только для выполнения операций записи. Если файл уже существовал, то он открывается также только для операции записи, при этом его длина уменьшается до 0 с одновременным сохранением всех других атрибутов файла.

Параметр `mode` устанавливает атрибуты прав доступа различных категорий пользователей к новому файлу при его создании. Этот параметр задается как сумма следующих восьмеричных значений:

0400 – разрешено чтение для пользователя, создавшего файл;
0200 – разрешена запись для пользователя, создавшего файл;
0100 – разрешено исполнение для пользователя, создавшего файл;
0040 – разрешено чтение для группы пользователя, создавшего файл;
0020 – разрешена запись для группы пользователя, создавшего файл;
0010 – разрешено исполнение для группы пользователя, создавшего файл;
0004 – разрешено чтение для всех остальных пользователей;
0002 – разрешена запись для всех остальных пользователей;
0001 – разрешено исполнение для всех остальных пользователей.

При создании файла реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса umask, а именно – они равны `mode & ~umask`.

Возвращаемое значение

Системный вызов возвращает значение файлового дескриптора для открытого файла при нормальном завершении и значение -1 при возникновении ошибки.

Операция чтения атрибутов файла. Системные вызовы `stat()`, `fstat()` и `lstat()`. Для чтения всех атрибутов файла в специальную структуру могут применяться системные вызовы `stat()`, `fstat()` и `lstat()`. Разъяснение понятий жесткой и мягкой (символической) связи, встречающихся в описании системных вызовов, будет дано позже при рассмотрении операций связывания файлов.

Системные вызовы для чтения атрибутов файла

Прототипы системных вызовов

```
#include <sys/stat.h>
#include <unistd.h>
int stat(char *filename, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(char *filename, struct stat *buf);
```

Описание системных вызовов

Настоящее описание не является полным описанием этих системных вызовов, а приспособлено для целей данного курса. Для получения полного описания обращайтесь в UNIX Manual.

Системные вызовы `stat`, `fstat` и `lstat` служат для получения информации об атрибутах файла.

Системный вызов `stat` читает информацию об атрибутах файла, на имя которого указывает параметр `filename`, и заполняет ими структуру, расположенную по адресу `buf`. Заметим, что имя файла должно быть полным, либо должно строиться относительно той директории, которая является текущей для процесса, совершившего вызов. Если имя файла относится к файлу типа «связь», то читается информация (рекурсивно!) об атрибутах файла, на который указывает символическая связь.

Системный вызов `lstat` идентичен системному вызову `stat` за одним исключением: если имя файла относится к файлу типа «связь», то читается информация о самом файле типа «связь».

Системный вызов `fstat` идентичен системному вызову `stat`, только файл задается не именем, а своим файловым дескриптором (естественно, файл к этому моменту должен быть открыт).

Для системных вызовов `stat` и `lstat` процессу не нужны никакие права доступа к указанному файлу, но могут понадобиться права для поиска во всех директориях, входящих в специфицированное имя файла.

Структура `stat` в различных версиях UNIX может быть описана по-разному. В Linux она содержит следующие поля:

```
struct stat {
dev_t st_dev; /* устройство, на котором расположен файл */
ino_t st_ino; /* номер индексного узла для файла */
mode_t st_mode; /* тип файла и права доступа к нему */
nlink_t st_nlink; /* счетчик числа жестких связей */
uid_t st_uid; /* идентификатор пользователя владельца */
gid_t st_gid; /* идентификатор группы владельца */
dev_t st_rdev; /* тип устройства для специальных файлов устройств */
off_t st_size; /* размер файла в байтах (если определен для данного
```

```
    типа файлов) */
  unsigbed long st_blksize; /* размер блока для файловой системы */
  unsigned long st_blocks; /* число выделенных блоков */
  time_t st_atime; /* время последнего доступа к файлу */
  time_t st_mtime; /* время последней модификации файла */
  time_t st_ctime; /* время создания файла */
}
```

Для определения типа файла можно использовать следующие логические макросы, применяя их к значению поля `st_mode`:

`S_ISLNK(m)` **– файл типа «связь»?**

`S_ISREG(m)` **– регулярный файл?**

`S_ISDIR(m)` **– директория?**

`S_ISCHR(m)` **– специальный файл символьного устройства?**

`S_ISBLK(m)` **– специальный файл блочного устройства?**

`S_ISFIFO(m)` **– файл типа FIFO?**

`S_ISSOCK(m)` **– файл типа «socket»?**

Младшие 9 бит поля `st_mode` **определяют права доступа к файлу подобно тому, как это делается в маске создания файлов текущего процесса.**

Возвращаемое значение

Системные вызовы возвращают значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Операции изменения атрибутов файла. Большинство операций изменения атрибутов файла обычно выполняется пользователем в интерактивном режиме с помощью команд операционной системы. О них уже шла речь в материалах семинаров 1–2, и мы не будем возвращаться к ним вновь. Отметим только операцию изменения размеров файла, а точнее операцию его обрезания, без изменения всех других атрибутов, кроме, быть может, времени последнего доступа к файлу и его последней модификации. Для того чтобы уменьшить размеры существующего файла до 0, не затрагивая остальных его характеристик (прав доступа, даты создания, учетной информации и т. д.), можно при открытии файла использовать в комбинации флагов системного вызова `open()` флаг `O_TRUNC`. Для изменения размеров файла до любой желаемой величины (даже для его увеличения во многих вариантах UNIX, хотя изначально этого не предусматривалось!) может использоваться системный вызов `ftruncate()`. При этом, если размер файла мы уменьшаем, то вся информация в конце файла, не влезающая в новый размер, будет потеряна. Если же размер файла мы увеличиваем, то это будет выглядеть так, как будто мы дополнили его до недостающего размера нулевыми байтами.

Системный вызов ftruncate()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
int ftruncate(int fd, size_t length);
```

Описание системного вызова

Системный вызов ftruncate предназначен для изменения длины открытого регулярного файла.

Параметр fd является дескриптором соответствующего файла, т. е. значением, которое вернул системный вызов open().

Параметр length – значение новой длины для этого файла. Если параметр length меньше, чем текущая длина файла, то вся информация в конце файла, не влезающая в новый размер, будет потеряна. Если же он` больше, чем текущая длина, то файл будет выглядеть так, как будто мы дополнили его до недостающего размера нулевыми байтами.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение –1 при возникновении ошибки.

Операции чтения из файла и записи в файл. Для операций чтения из файла и записи в файл применяются системные вызовы read() и write(), которые мы уже обсуждали ранее (семинар 5, раздел «Системные вызовы read(), write(), close()»).

Надо отметить, что их поведение при работе с файлами имеет определенные особенности, связанные с понятием указателя текущей позиции в файле.

При работе с файлами информация записывается в файл или читается из него, начиная с места, определяемого указателем текущей позиции в файле. Значение указателя увеличивается на количество реально прочитанных или записанных байт. При чтении информации из файла она не пропадает из него. Если системный вызов read возвращает значение 0, то это означает, что достигнут конец файла.

Операция изменения указателя текущей позиции. Системный вызов lseek(). С точки зрения процесса все регулярные файлы являются файлами прямого доступа. В любой момент процесс может изменить положение указателя текущей позиции в открытом файле с помощью системного вызова lseek().

Особенностью этого системного вызова является возможность помещения указателя текущей позиции в файле за конец файла (т. е. возможность установления значения указателя большего, чем длина файла).

При любой последующей операции записи в таком положении указателя файл будет выглядеть так, как будто возникший промежуток от конца файла до текущей позиции, где начинается запись, был заполнен нулевыми байтами. Если операции записи в таком положении указателя не производится, то никакого изменения файла, связанного с необычным значением указателя, не произойдет (например, операция чтения будет возвращать нулевое значение для количества прочитанных байтов).

Системный вызов lseek()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

Описание системного вызова

Системный вызов lseek предназначен для изменения положения указателя текущей позиции в открытом регулярном файле.

Параметр fd является дескриптором соответствующего файла, т. е. значением, которое вернул системный вызов open().

Параметр offset совместно с параметром whence определяют новое положение указателя текущей позиции следующим образом:

- Если значение параметра whence равно SEEK_SET, то новое значение указателя будет составлять offset байт от начала файла. Естественно, что значение offset в этом случае должно быть не отрицательным.
- значение параметра whence равно SEEK_CUR, то новое значение указателя будет составлять старое значение указателя + offset байт. При этом новое значение указателя не должно стать отрицательным.
- Если значение параметра whence равно SEEK_END, то новое значение указателя будет составлять длина файла + offset байт. При этом новое значение указателя не должно стать отрицательным.

Системный вызов lseek позволяет выставить текущее значение указателя за конец файла (т. е. сделать его превышающим размер файла). При любой последующей операции записи в этом положении указателя файл будет выглядеть так, как будто возникший промежуток был заполнен нулевыми битами.

Тип данных off_t обычно является синонимом типа long.

Возвращаемое значение

Системный вызов возвращает новое положение указателя текущей позиции в байтах от начала файла при нормальном завершении и значение -1 при возникновении ошибки.

Операция добавления информации в файл. Флаг O_APPEND. Хотя эта операция по сути дела является комбинацией двух уже рассмотренных операций, мы считаем нужным упомянуть ее особо. Если открытие файла системным вызовом `open()` производилось с установленным флагом `O_APPEND`, то любая операция записи в файл **будет всегда добавлять новые данные в конец файла**, независимо от предыдущего положения указателя текущей позиции (как если бы непосредственно перед записью был выполнен вызов `lseek()` для установки указателя на конец файла).

Операции создания связей. Команда `ln`, системные вызовы `link()` и `symlink()`. С операциями, позволяющими изменять логическую структуру файловой системы, такими как создание файла, мы уже сталкивались в этом разделе. Однако операции создания связи служат для проведения новых именованных ребер в уже существующей структуре без добавления новых узлов или для опосредованного проведения именованного ребра к уже существующему узлу через файл типа «связь» и неименованное ребро. Такие операции мы до сих пор не рассматривали, поэтому давайте остановимся на них подробнее.

Допустим, что несколько программистов совместно ведут работу над одним и тем же проектом. Файлы, относящиеся к этому проекту, вполне естественно могут быть выделены в отдельную директорию так, чтобы не смешиваться с файлами других пользователей и другими файлами программистов, участвующих в проекте. Для удобства каждый из разработчиков, конечно, хотел бы, чтобы эти файлы находились в его собственной директории. Этого можно было бы добиться, копируя по мере изменения новые версии соответствующих файлов из директории одного исполнителя в директорию другого исполнителя. Однако тогда, во-первых, возникнет ненужное дублирование информации на диске. Во-вторых, появится необходимость решения тяжелой задачи: синхронизации обновления замены всех копий этих файлов новыми версиями.

Существует другое решение проблемы. Достаточно разрешить файлам иметь несколько имен. Тогда одному физическому экземпляру данных на диске могут соответствовать различные имена файла, находящиеся в одной или в разных директориях. Подобная операция присвоения нового имени файлу (без уничтожения ранее существовавшего имени) получила название операции создания связи.

В операционной системе UNIX связь может быть создана двумя различными способами.

Первый способ, наиболее точно следующий описанной выше процедуре, получил название способа создания жесткой связи (hard link). С точки зрения логической структуры файловой системы этому способу соответствует проведение нового именованного ребра из узла, соответствующего некоторой директории, к узлу, соответствующему файлу любого

типа, получающему дополнительное имя. С точки зрения структур данных, описывающих строение файловой системы, в эту директорию добавляется запись, содержащая дополнительное имя файла и номер его индексного узла (уже существующий!). При таком подходе и новое имя файла, и его старое имя или имена абсолютно равноправны для операционной системы и могут взаимозаменяемо использоваться для осуществления всех операций.

Использование жестких связей приводит к возникновению двух проблем.

Первая проблема связана с операцией удаления файла. Если мы хотим удалить файл из некоторой директории, то после удаления из ее содержимого записи, соответствующей этому файлу, мы не можем освободить логические блоки, занимаемые файлом, и его индексный узел, не убедившись, что у файла нет дополнительных имен (к его индексному узлу не ведут ссылки из других директорий), иначе мы нарушим целостность файловой системы. Для решения этой проблемы файлы получают дополнительный атрибут – счетчик жестких связей (или именованных ребер), ведущих к ним, который, как и другие атрибуты, располагается в их индексных узлах. При создании файла этот счетчик получает значение 1. При создании каждой новой жесткой связи, ведущей к файлу, он увеличивается на 1. Когда мы удаляем файл из некоторой директории, то из ее содержимого удаляется запись об этом файле, и счетчик жестких связей уменьшается на 1. Если его значение становится равным 0, происходит освобождение логических блоков и индексного узла, выделенных этому файлу.

Вторая проблема связана с опасностью превращения логической структуры файловой системы из ациклического графа в циклический и с возможной неопределенностью толкования записи с именем «..» в содержимом директорий. Для их предотвращения во всех существующих вариантах операционной системы UNIX запрещено создание жестких связей, ведущих к уже существующим директориям (несмотря на то, что POSIX-стандарт для операционной системы UNIX разрешает подобную операцию для пользователя root). Поэтому мы и говорили о том, что в узел, соответствующий файлу типа «директория», не может вести более одного именованного ребра. (В операционной системе Linux по непонятной причине дополнительно запрещено создание жестких связей, ведущих к специальным файлам устройств.)

Команда ln

Синтаксис команды

```
ln [options] source [dest]
ln [options] source ... directory
```

Описание команды

Настоящее описание не является полным описанием команды `ln`, а описывает только ее опции, используемые в данном курсе. Для получения полного описания обращайтесь к UNIX Manual.

Команда `ln` предназначена для реализации операции создания связи в файловой системе. В нашем курсе мы будем использовать две формы этой команды.

Первая форма команды, когда в качестве параметра `source` задается имя только одного файла, а параметр `dest` отсутствует, или когда в качестве параметра `dest` задается имя файла, не существующего в файловой системе, создает связь к файлу, указанному в качестве параметра source, в текущей директории с его именем (если параметр `dest` отсутствует) или с именем `dest` (полным или относительным) в случае наличия параметра `dest`.

Вторая форма команды, когда в качестве параметра `source` задаются имена одного или нескольких файлов, разделенные между собой пробелами, а в качестве параметра `directory` задается имя уже существующей в файловой системе директории, создает связи к каждому из файлов, перечисленных в параметре `source`, в директории `directory` с именами, совпадающими с именами перечисленных файлов.

Команда `ln` без опций служит для создания жестких связей (hard link), а команда `ln` с опцией `-s` – для создания мягких (soft link) или символических (symbolic) связей.

Примечание: во всех существующих версиях UNIX (несмотря на стандарт POSIX) запрещено создание жестких связей к директориям. Операционная система Linux запрещает также, по непонятным причинам, создание жестких связей к специальным файлам устройств.

Для создания жестких связей применяются команда операционной системы `ln` без опций и системный вызов `link()`.

Надо отметить, что системный вызов `link()` является одним из немногих системных вызовов, совершающих операции над файлами, которые не требуют предварительного открытия файла, поскольку он подразумевает выполнение единичного действия только над содержимым индексного узла, выделенного связываемому файлу.

Системный вызов link()

Прототип системного вызова

```
#include <unistd.h>
int link(char *pathname, char *linkpathname);
```

Описание системного вызова

Системный вызов `link` служит для создания жесткой связи к файлу с именем, на которое указывает параметр `pathname`. Указатель на имя создаваемой связи задается параметром `linkpathname` (полное или относительное имя связи). Во всех существующих реализациях операционной системы UNIX запрещено создавать жесткие связи к директориям. В операционной

системе Linux (по непонятной причине) дополнительно запрещено создавать жесткие связи к специальным файлам устройств.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Второй способ создания связи получил название способа создания *мягкой* (*soft*) или *символической* (*symbolic*) *связи* (*link*). В то время как жесткая связь файлов является аналогом использования прямых ссылок (указателей) в современных языках программирования, символическая связь, до некоторой степени, напоминает косвенные ссылки (указатель на указатель). При создании мягкой связи с именем `symlink` из некоторой директории к файлу, заданному полным или относительным именем `linkpath`, в этой директории действительно **создается новый файл типа «связь»** с именем `symlink` со своими собственными индексным узлом и логическими блоками. При тщательном рассмотрении можно обнаружить, что все его содержимое составляет только символьная запись имени `linkpath`. Операция открытия файла типа «связь» устроена таким образом, что в действительности открывается не сам этот файл, а тот файл, чье имя содержится в нем (при необходимости рекурсивно!). Поэтому операции над файлами, требующие предварительного открытия файла (как, впрочем, и большинство команд операционной системы, совершающих действия над файлами, где операция открытия файла присутствует, но скрыта от пользователя), в реальности будут совершаться не над файлом типа «связь», а над тем файлом, имя которого содержится в нем (или над тем файлом, который, в конце концов, откроется при рекурсивных ссылках). Отсюда, в частности, следует, что попытки прочитать **реальное** содержимое файлов типа «связь» с помощью системного вызова `read()` обречены на неудачу. Как видно, создание мягкой связи, с точки зрения изменения логической структуры файловой системы, эквивалентно опосредованному проведению именованного ребра к уже существующему узлу через файл типа «связь» и неименованное ребро.

Создание символической связи не приводит к проблеме, связанной с удалением файлов. Если файл, на который ссылается мягкая связь, удаляется с физического носителя, то попытка открытия файла мягкой связи (а, следовательно, и удаленного файла) приведет к ошибке «Файла с таким именем не существует», которая может быть аккуратно обработана приложением. Таким образом, удаление связанного объекта, как упоминалось ранее, лишь отчасти и не фатально нарушит целостность файловой системы.

Неаккуратное применение символических связей пользователями операционной системы может привести к превращению логической структуры файловой системы из ациклического графа в циклический граф. Это, конечно, нежелательно, но не носит столь разрушительного характера, как циклы, которые могли бы быть созданы жесткой связью, если бы не был введен запрет на организацию жестких связей к директориям. Поскольку мягкие связи принципиально отличаются от жестких связей и связей, возникающих между директорией и файлом при его создании, мягкая связь легко может быть идентифицирована операционной системой или программой пользователя. Для предотвращения зацикливания программ, выполняющих операции над файлами, обычно ограничивается глубина рекурсии по прохождению мягких связей. Превышение этой глубины приводит к возникновению ошибки «Слишком много мягких связей», которая может быть легко обработана приложением. Поэтому ограничения на тип файлов, к которым может вести мягкая связь, в операционной системе UNIX не вводятся.

Для создания мягких связей применяются уже знакомая нам команда операционной системы `ln` с опцией `-s` и системный вызов `symlink()`. Надо отметить, что системный вызов `symlink()` также не требует предварительного открытия связываемого файла, поскольку он вообще не рассматривает его содержимое.

Системный вызов symlink()

Прототип системного вызова

```
#include <unistd.h>
int symlink(char *pathname, char *linkpathname);
```

Описание системного вызова

Системный вызов `symlink` служит для создания символической (мягкой) связи к файлу с именем, на которое указывает параметр `pathname`. Указатель на имя создаваемой связи задается параметром `linkpathname` (полное или относительное имя связи).

Никакой проверки реального существования файла с именем `pathname` системный вызов не производит.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Операция удаления связей и файлов. Системный вызов `unlink()`. При рассмотрении операции связывания файлов мы уже почти полностью рассмотрели, как производится операция удаления жестких связей и файлов.

При удалении мягкой связи, т. е. фактически файла типа «связь», все происходит как и для обычных файлов. Единственным изменением, с точки зрения логической структуры файловой системы, является то, что при действительном удалении узла, соответствующего файлу типа «связь», вместе с ним удаляется и выходящее из него неименованное ребро.

Дополнительно необходимо отметить, что условием реального удаления регулярного файла с диска является не только равенство 0 значения его счетчика жестких связей, но и отсутствие процессов, которые держат этот файл открытым. Если такие процессы есть, то удаление регулярного файла будет выполнено при его полном закрытии последним использующим файл процессом.

Для осуществления операции удаления жестких связей и/или файлов можно задействовать уже известную вам из семинаров 1–2 команду операционной системы `rm` или системный вызов `unlink()`.

Заметим, что системный вызов `unlink()` также не требует предварительного открытия удаляемого файла, поскольку после его удаления совершать над ним операции бессмысленно.

Системный вызов unlink()

Прототип системного вызова

```
#include <unistd.h>
int unlink(char *pathname);
```

Описание системного вызова

Системный вызов `unlink` служит для удаления имени, на которое указывает параметр `pathname`, из файловой системы.

Если после удаления имени счетчик числа жестких связей у данного файла стал равным 0, то возможны следующие ситуации:

- Если в операционной системе нет процессов, которые держат данный файл открытым, то файл полностью удаляется с физического носителя.
- Если удаляемое имя было последней жесткой связью для регулярного файла, но какой-либо процесс держит его открытым, то файл продолжает существовать до тех пор, пока не будет закрыт последний файловый дескриптор, ссылающийся на данный файл.
- Если имя относится к файлу типа socket, FIFO или к специальному файлу устройства, то файл удаляется независимо от наличия процессов, держащих его открытым, но процессы, открывшие данный объект, могут продолжать пользоваться им.
- Если имя относится к файлу типа «связь», то он удаляется, и мягкая связь оказывается разорванной.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Практическое применение команд и системных вызовов для операций над файлами

Практическое использование изученных вызовов и команд обычно проводится по усмотрению преподавателя или собственному вашему усмотрению. Создайте жесткие и символические связи из вашей директории к другим файлам. Просмотрите содержимое директорий со связями с помощью команды `ls -al`. Обратите внимание на отличие мягких и жестких связей в листинге этой команды. Определите допустимую глубину рекурсии символических связей для вашей операционной системы.

Специальные функции для работы с содержимым директорий

Стандартные системные вызовы `open()`, `read()` и `close()` не могут помочь программисту изучить содержимое файла типа «директория». Для анализа содержимого директорий используется набор функций из стандартной библиотеки языка C.

Функция opendir()

Прототип функции

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(char *name);
```

Описание функции

Функция `opendir` служит для открытия потока информации для директории, имя которой расположено по указателю `name`. Тип данных `DIR` представляет собой некоторую структуру данных, описывающую такой поток. Функция `opendir` подготавливает почву для функционирования других функций, выполняющих операции над директорией, и позиционирует поток на первой записи директории.

Возвращаемое значение

При удачном завершении функция возвращает указатель на открытый поток директории, который будет в дальнейшем передаваться в качестве параметра всем другим функциям, работающим с этой директорией. При неудачном завершении возвращается значение `NULL`.

С точки зрения программиста в этом интерфейсе директория представляется как файл последовательного доступа, над которым можно совершать операции чтения очередной записи и позиционирования на на-

чале файла. Перед выполнением этих операций директорию необходимо открыть, а после окончания – закрыть. Для открытия директории используется функция `opendir()`, которая подготавливает почву для совершения операций и позиционирует нас на начале файла. Чтение очередной записи из директории осуществляет функция `readdir()`, одновременно позиционируя нас на начале следующей записи (если она, конечно, существует). Для операции нового позиционирования на начале директории (если вдруг понадобится) применяется функция `rewinddir()`. После окончания работы с директорией ее необходимо закрыть с помощью функции `closedir()`.

Функция readdir()

Прототип функции

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

Описание функции

Функция `readdir` служит для чтения очередной записи из потока информации для директории.

Параметр `dir` представляет собой указатель на структуру, описывающую поток директории, который вернула функция `opendir()`.

Тип данных `struct dirent` представляет собой некоторую структуру данных, описывающую одну запись в директории. Поля этой записи сильно варьируются от одной файловой системы к другой, но одно из полей, которое собственно и будет нас интересовать, всегда присутствует в ней. Это поле `char d_name[ ]` неопределенной длины, не превышающей значения `NAME_MAX+1`, которое содержит символьное имя файла, завершающееся символом конца строки. Данные, возвращаемые функцией `readdir`, переписываются при очередном вызове этой функции для того же самого потока директории.

Возвращаемое значение

При удачном завершении функция возвращает указатель на структуру, содержащую очередную запись директории. При неудачном завершении или при достижении конца директории возвращается значение `NULL`.

Функция rewinddir()

Прототип функции

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir(DIR *dir);
```

Описание функции

Функция `rewinddir` служит для позиционирования потока информации для директории, ассоциированного с указателем `dir` (т. е. с тем, который вернула функция `opendir()`), на первой записи (или на начале) директории.

Функция closedir()

Прототип функции

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dir);
```

Описание функции

Функция `closedir` служит для закрытия потока информации для директории, ассоциированного с указателем `dir` (т. е. с тем, который вернула функция `opendir()`). После закрытия поток директории становится недоступным для дальнейшего использования.

Возвращаемое значение

При успешном завершении функция возвращает значение `0`, при неудачном завершении – значение `-1`.

Написание, прогон и компиляция программы, анализирующей содержимое директории

Напишите, откомпилируйте и прогоните программу, распечатывающую список файлов, входящих в директорию, с указанием их типов. Имя директории задается как параметр командной строки. Если оно отсутствует, то выбирается текущая директория.

Задача повышенной сложности: напишите программу, распечатывающую содержимое заданной директории в формате, аналогичном формату выдачи команды `ls -al`. Для этого вам дополнительно понадобится са-

мостоятельно изучить в UNIX Manual функцию `ctime(3)` и системные вызовы `time(2)`, `readlink(2)`. Цифры после имен функций и системных вызовов – это номера соответствующих разделов для UNIX Manual.

Понятие о файлах, отображаемых в память (memory mapped файлах). Системные вызовы mmap(), munmap()

Как уже говорилось, с помощью системного вызова `open()` операционная система отображает файл из пространства имен в дисковое пространство файловой системы, подготавливая почву для осуществления других операций. С появлением концепции виртуальной памяти, которая рассматривалась в лекции 9, когда физические размеры памяти перестали играть роль сдерживающего фактора в развитии вычислительных систем, стало возможным отображать файлы непосредственно в адресное пространство процессов. Иными словами, появилась возможность работать с файлами как с обычной памятью, заменив выполнение базовых операций над ними с помощью системных вызовов на использование операций обычных языков программирования. Файлы, чье содержимое отображается непосредственно в адресное пространство процессов, получили название файлов, отображаемых в память, или, по-английски, memory mapped файлов (см. лекцию 10). Надо отметить, что такое отображение может быть осуществлено не только для всего файла в целом, но и для его части.

С точки зрения программиста работа с такими файлами выглядит следующим образом:

1. Отображение файла из пространства имен в адресное пространство процесса происходит в два этапа: сначала выполняется отображение в дисковое пространство, а уже затем из дискового пространства в адресное. Поэтому вначале файл необходимо открыть, используя обычный системный вызов `open()`.
2. Вторым этапом является отображение файла целиком или частично из дискового пространства в адресное пространство процесса. Для этого используется системный вызов `mmap()`. Файл после этого можно и закрыть, выполнив системный вызов `close()`, так как необходимую информацию о расположении файла на диске мы уже сохранили в других структурах данных при вызове `mmap()`.

Системный вызов mmap()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
void *mmap (void *start, size_t length, int prot, int flags, int fd,
   off_t offset);
```

Описание системного вызова

Системный вызов mmap служит для отображения предварительно открытого файла (например, с помощью системного вызова open()) в адресное пространство вычислительной системы. После его выполнения файл может быть закрыт (например, системным вызовом close()), что никак не повлияет на дальнейшую работу с отображенным файлом.

Настоящее описание не является полным описанием системного вызова, а предназначено только для использования в рамках данного курса. Для получения полной информации обращайтесь к UNIX Manual.

Параметр fd является файловым дескриптором для файла, который мы хотим отобразить в адресное пространство (т. е. значением, которое вернул системный вызов open()).

Ненулевое значение параметра start может использоваться только очень квалифицированными системными программистами, поэтому мы в семинарах будем всегда полагать его равным значению NULL, позволяя операционной системе самой выбрать начало области адресного пространства, в которую будет отображен файл.

В память будет отображаться часть файла, начиная с позиции внутри его, заданной значением параметра offset – смещение от начала файла в байтах, и длиной, равной значению параметра length (естественно, тоже в байтах). Значение параметра length может и превышать реальную длину от позиции offset до конца существующего файла. На поведении системного вызова это никак не отразится, но в дальнейшем при попытке доступа к ячейкам памяти, лежащим вне границ реального файла, возникнет сигнал SIGBUS (реакция на него по умолчанию – прекращение процесса с образованием core файла).

Параметр flags определяет способ отображения файла в адресное пространство. В рамках нашего курса мы будем использовать только два его возможных значения: MAP_SHARED и MAP_PRIVATE. Если в качестве его значения выбрано MAP_SHARED, то полученное отображение файла впоследствии будет использоваться и другими процессами, вызвавшими mmap для этого файла с аналогичными значениями параметров, а все изменения, сделанные в отображенном файле, будут сохранены во вторичной памяти. Если в качестве значения параметра flags указано MAP_PRIVATE, то процесс получает отображение файла в свое монопольное распоряжение, но все изменения в нем не могут быть занесены во вторичную память (т. е., проще говоря, не сохранятся).

Параметр prot определяет разрешенные операции над областью памяти, в которую будет отображен файл. В качестве его значения мы будем использовать значения PROT_READ (разре-

шено чтение), PROT_WRITE (разрешена запись) или их комбинацию через операцию «побитовое или» – «|». Необходимо отметить две существенные особенности системного вызова, связанные с этим параметром:

1. Значение параметра prot не может быть шире, чем операции над файлом, заявленные при его открытии в параметре flags системного вызова open(). Например, нельзя открыть файл только для чтения, а при его отображении в память использовать значение prot = PROT_READ | PROT_WRITE.
2. В результате ошибки в операционной системе Linux при работе на 486-х и 586-х процессорах попытка записать в отображение файла, открытое только для записи, более 32-х байт одновременно приводит к ошибке (возникает сигнал о нарушении защиты памяти).

Возвращаемое значение

При нормальном завершении системный вызов возвращает начальный адрес области памяти, в которую отображен файл (или его часть), при возникновении ошибки – специальное значение MAP_FAILED.

3. После этого с содержимым файла можно работать, как с содержимым обычной области памяти.
4. По окончании работы с содержимым файла необходимо освободить дополнительно выделенную процессу область памяти, предварительно синхронизировав содержимое файла на диске с содержимым этой области (если это, конечно, необходимо). Эти действия выполняет системный вызов munmap().

Системный вызов munmap

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
int munmap (void *start, size_t length);
```

Описание системного вызова

Системный вызов munmap служит для прекращения отображения memory mapped файла в адресное пространство вычислительной системы. Если при системном вызове mmap() было задано значение параметра flags, равное MAP_SHARED, и в отображении файла была разрешена операция записи (в параметре prot использовалось значение PROT_WRITE), то munmap синхронизирует содержимое отображения с содержимым файла во вторичной памяти. После его выполнения области памяти, использовавшиеся для отображения файла, становятся недоступны текущему процессу.

Параметр `start` является адресом начала области памяти, выделенной для отображения файла, т. е. значением, которое вернул системный вызов `mmap()`.

Параметр `length` определяет ее длину, и его значение должно совпадать со значением соответствующего параметра в системном вызове `mmap()`.

Возвращаемое значение

При нормальном завершении системный вызов возвращает значение `0`, при возникновении ошибки – значение `-1`.

Анализ, компиляция и прогон программы для создания memory mapped файла и записи его содержимого

Для закрепления материала, изложенного в предыдущем разделе, рассмотрим пример программы:

```
/* Программа 11-1.c для иллюстрации работы с
memory mapped файлом */
int main(void)
{
  int fd; /* Файловый дескриптор для файла, в
  котором будет храниться наша информация*/
  size_t length; /* Длина отображаемой части файла */
  int i;
  /* Ниже следует описание типа структуры, которым мы
  забьем файл, и двух указателей на подобный тип.
  Указатель ptr будет использоваться в качестве
  начального адреса выделенной области памяти,
  а указатель tmpptr - для перемещения внутри этой
  области. */
  struct A {
    double f;
    double f2;
  } *ptr, tmpptr;
  /* Открываем файл или сначала создаем его (если
  такого файла не было). Права доступа к файлу при
  создании определяем как read and write для всех
  категорий пользователей (0666). Из-за ошибки в
  Linux мы будем вынуждены ниже в системном вызове
  mmap() разрешить в отображении файла и чтение, и
  запись, хотя реально нам нужна только запись.
```

```
Поэтому и при открытии файла мы вынуждены задавать
O_RDWR. */
fd = open("mapped.dat", O_RDWR | O_CREAT, 0666);
if( fd == -1){
  /* Если файл открыть не удалось, выдаем
  сообщение об ошибке и завершаем работу */
  printf("File open failed!\n");
  exit(1);
}
/* Вычисляем будущую длину файла (мы собираемся
записать в него 100000 структур) */
length = 100000*sizeof(struct A);
/* Вновь созданный файл имеет длину 0. Если мы
его отобразим в память с такой длиной, то любая
попытка записи в выделенную память приведет к
ошибке. Увеличиваем длину файла с помощью вызова
ftruncate(). */
ftruncate(fd,length);
/* Отображаем файл в память. Разрешенные операции
над отображением указываем как PROT_WRITE |
PROT_READ по уже названным причинам. Значение
флагов ставим в MAP_SHARED, так как мы хотим
сохранить информацию, которую занесем в отображение,
на диске. Файл отображаем с его начала
(offset = 0) и до конца (length = длине файла). */
ptr = (struct A )mmap(NULL, length, PROT_WRITE |
PROT_READ, MAP_SHARED, fd, 0);
/* Файловый дескриптор нам более не нужен, и мы
его закрываем */
close(fd);
if( ptr == MAP_FAILED ){
  /* Если отобразить файл не удалось, сообщаем
  об ошибке и завершаем работу */
  printf("Mapping failed!\n");
  exit(2);
}
/* В цикле заполняем образ файла числами от 1 до
100000 и их квадратами. Для перемещения по области
памяти используем указатель tmpptr, так как
указатель ptr на начало образа файла нам понадобится
для прекращения отображения вызовом munmap(). */
tmpptr = ptr;
```

```
    for(i = 1; i <=100000; i++){
        tmpptr->f = i;
        tmpptr->f2 = tmpptr->f*tmpptr->f;
        tmpptr++;
    }
    /* Прекращаем отображать файл в память, записываем
    содержимое отображения на диск и освобождаем память. */
    munmap((void *)ptr, length);
    return 0;
}
```

Эта программа создает файл, отображает его в адресное пространство процесса и заносит в него информацию с помощью обычных операций языка С.

Обратите внимание на необходимость увеличения размера файла перед его отображением. Созданный файл имеет нулевой размер, и если его с этим размером отобразить в память, то мы сможем записать в него или прочитать из него не более 0 байт, т. е. ничего. Для увеличения размера файла использован системный вызов `ftruncate()`, хотя это можно было бы сделать и любым другим способом.

При отображении файла мы вынуждены разрешить в нем и запись, и чтение, хотя реально совершаем только запись. Это сделано для того, чтобы избежать ошибки в операционной системе Linux, связанной с использованием 486-х и 586-х процессоров. Такой список разрешенных операций однозначно требует, чтобы при открытии файла системным вызовом `open()` файл открывался и на запись, и на чтение. Поскольку информацию мы желаем сохранить на диске, при отображении использовано значение флагов `MAP_SHARED`. Откомпилируйте эту программу и запустите ее.

Изменение предыдущей программы для чтения из файла, используя его отображение в память

Модифицируйте программу из предыдущего раздела так, чтобы она отображала файл, записанный программой из раздела «Анализ, компиляция и прогон программы для создания memory mapped файла и записи его содержимого», в память и считала сумму квадратов чисел от 1 до 100000, которые уже находятся в этом файле.

Задача повышенной сложности: напишите две программы, использующие memory mapped файл для обмена информацией при одновременной работе, подобно тому, как они могли бы использовать разделяемую память.

Семинары 12–13. Организация ввода-вывода в UNIX. Файлы устройств. Аппарат прерываний. Сигналы в UNIX

Понятие виртуальной файловой системы. Операции над файловыми системами. Монтирование файловых систем. Блочные, символьные устройства. Понятие драйвера. Блочные, символьные драйверы, драйверы низкого уровня. Файловый интерфейс. Аппаратные прерывания (interrupt), исключения (exception), программные прерывания (trap, software interrupt). Их обработка. Понятие сигнала. Способы возникновения сигналов и виды их обработки. Понятия группы процессов, сеанса, лидера группы, лидера сеанса, управляющего терминала сеанса. Системные вызовы `getpgrp()`, `setpgrp()`, `getpgid()`, `setpgid()`, `getsid()`, `setsid()`. Системный вызов `kill()` и команда `kill()`. Системный вызов `signal()`. Установка собственного обработчика сигнала. Восстановление предыдущей реакции на сигнал. Сигналы `SIGUSR1` и `SIGUSR2`. Использование сигналов для синхронизации процессов. Завершение порожденного процесса. Системный вызов `waitpid()`. Сигнал `SIGCHLD`. Возникновение сигнала `SIGPIPE` при попытке записи в pipe или FIFO, который никто не собирается читать. Понятие надежности сигналов. POSIX-функции для работы с сигналами.

Ключевые слова: виртуальная файловая система, виртуальный узел (vnode), таблица виртуальных узлов открытых файлов, таблица операций, монтирование файловых систем, команды `mount` и `umount`, блочные и символьные устройства, драйверы устройств, коммутатор устройств, старший и младший номера устройств, аппаратные прерывания, исключения, программные прерывания, сигналы, группа процессов, сеанс, лидер группы процессов, лидер сеанса, системные вызовы `getpgrp()`, `setpgrp()`, `getpgid()`, `setpgid()`, `getsid()`, `setsid()`, управляющий терминал сеанса, текущая группа процессов, фоновая группа процессов, эффективный идентификатор пользователя (EUID), команда `kill`, системные вызовы `kill()`, `signal()`, сигналы `SIGHUP`, `SIGQUIT`, `SIGINT`, `SIGCHLD`, `SIGPIPE`, `SIGUSR1`, `SIGUSR2`, `SIGKILL`, `SIGTERM`, системные вызовы `waitpid()` и `wait()`, POSIX-сигналы.

Понятие виртуальной файловой системы

Семинары 10–11 были посвящены устройству файловой системы s5fs. Существуют и другие файловые системы, имеющие архитектуру, отличную от архитектуры s5fs (иные способы отображения файла на пространство

физического носителя, иное построение директорий и т. д.). Современные версии UNIX-подобных операционных систем умеют работать с разнообразными файловыми системами, различающимися своей организацией. Такая возможность достигается с помощью разбиения каждой файловой системы на зависимую и независимую от конкретной реализации части, подобно тому, как в лекции 13, посвященной вопросам ввода-вывода, мы отделяли аппаратно-зависимые части для каждого устройства – драйверы – от общей базовой подсистемы ввода-вывода. Независимые части всех файловых систем одинаковы и представляют для всех остальных элементов ядра абстрактную файловую систему, которую принято называть виртуальной файловой системой. Зависимые части для различных файловых систем могут встраиваться в ядро на этапе компиляции либо добавляться к нему динамически по мере необходимости, без перекомпиляции системы (как в системах с микроядерной архитектурой).

Рассмотрим схематично устройство виртуальной файловой системы. В файловой системе s5fs данные о физическом расположении и атрибутах каждого открытого файла представлялись в операционной системе структурой данных в таблице индексных узлов открытых файлов (см. семинар 10–11, раздел «Системные вызовы и команды для выполнения операций над файлами и директориями»), содержащей информацию из индексного узла файла во вторичной памяти. В виртуальной файловой системе, в отличие от s5fs, каждый файл характеризуется не индексным узлом inode, а некоторым виртуальным узлом vnode. Соответственно, вместо таблицы индексных узлов открытых файлов в операционной системе появляется *таблица виртуальных узлов открытых файлов*. При открытии файла в операционной системе для него заполняется (если, конечно, не был заполнен раньше) элемент таблицы виртуальных узлов открытых файлов, в котором хранятся, как минимум, тип файла, счетчик числа открытий файла, **указатель** на реальные физические данные файла и, **обязательно, указатель** на таблицу системных вызовов, совершающих операции над файлом, – *таблицу операций*. Реальные физические данные файла (равно как и способ расположения файла на диске и т. п.) и системные вызовы, реально выполняющие операции над файлом, уже не являются элементами виртуальной файловой системы. Они относятся к одной из зависимых частей файловой системы, так как определяются ее конкретной реализацией.

При выполнении операций над файлами по таблице операций, чей адрес содержится в vnode, определяется системный вызов, который будет на самом деле выполнен над реальными физическими данными файла, чей адрес также находится в vnode. В случае с s5fs данные, на которые ссылается vnode, – это как раз данные индексного узла, рассмотренные на семинарах 10–11 и на лекции 12. Заметим, что таблица операций является общей для всех файлов, принадлежащих одной и той же файловой системе.

Операции над файловыми системами. Монтирование файловых систем

В материалах семинаров 11–12 рассматривалась только одна файловая система, расположенная в одном разделе физического носителя. Как только мы переходим к сосуществованию нескольких файловых систем в рамках одной операционной системы, встает вопрос о логическом объединении структур этих файловых систем. При работе операционной системы нам изначально доступна лишь одна, так называемая корневая, файловая система. Прежде, чем приступить к работе с файлом, лежащим в некоторой другой файловой системе, мы должны встроить ее в уже существующий ациклический граф файлов. Эта операция – операция над файловой системой – называется монтированием файловой системы (mount).

Для монтирования файловой системы (см. лекцию 12, раздел «Монтирование файловых систем») в существующем графе должна быть найдена или создана некоторая пустая директория – точка монтирования, к которой и присоединится корень монтируемой файловой системы. При операции монтирования в ядре заводятся структуры данных, описывающие файловую систему, а в vnode для точки монтирования файловой системы помещается специальная информация.

Монтирование файловых систем обычно является прерогативой системного администратора и осуществляется командой операционной системы mount в ручном режиме, либо автоматически при старте операционной системы. Использование этой команды без параметров не требует специальных полномочий и позволяет пользователю получить информацию обо всех смонтированных файловых системах и соответствующих им физических устройствах. Для пользователя также обычно разрешается монтирование файловых систем, расположенных на гибких магнитных дисках. Для первого накопителя на гибких магнитных дисках такая команда в Linux будет выглядеть следующим образом:

```
mount /dev/fd0 <имя пустой директории>
```

где <имя пустой директории> описывает точку монтирования, а /dev/fd0 – специальный файл устройства, соответствующего этому накопителю (о специальных файлах устройств будет подробно рассказано в следующем разделе).

Команда mount

Синтаксис команды

```
mount [-hV]
mount [-rw] [-t fstype] device dir
```

Описание команды

Настоящее описание не является полным описанием команды `mount`, а описывает только те ее опции (очень малую часть), которые используются в данном курсе. Для получения полного описания следует обратиться к UNIX Manual.

Команда `mount` предназначена для выполнения операции монтирования файловой системы и получения информации об уже смонтированных файловых системах.

Опции `-h`, `-V` используются при вызове команды без параметров и служат для следующих целей:

`-h` – вывести краткую инструкцию по пользованию командой;

`-V` – вывести информацию о версии команды `mount`.

Команда `mount` без опций и без параметров выводит информацию обо всех уже смонтированных файловых системах.

Команда `mount` с параметрами служит для выполнения операции монтирования файловой системы.

Параметр `device` задает имя специального файла для устройства, содержащего файловую систему.

Параметр `dir` задает имя точки монтирования (имя некоторой уже существующей пустой директории). При монтировании могут использоваться следующие опции:

`-r` – смонтировать файловую систему только для чтения (read only);

`-w` – смонтировать файловую систему для чтения и для записи (read/write). Используется по умолчанию;

`-t fstype` – задать тип монтируемой файловой системы как `fstype`. Поддерживаемые типы файловых систем в операционной системе Linux: adfs, affs, autofs, coda, coherent, cramfs, devpts, efs, ext, ext2, ext3, hfs, hpfs, iso9660 (для CD), minix, msdos, ncpfs, nfs, ntfs, proc, qnx4, reiserfs, romfs, smbfs, sysv, udf, ufs, umsdos, vfat, xenix, xfs, xiafs. При отсутствии явно заданного типа команда для большинства типов файловых систем способна опознать его автоматически.

Если мы не собираемся использовать смонтированную файловую систему в дальнейшем (например, хотим вынуть ранее смонтированную дискету), нам необходимо выполнить операцию логического разъединения смонтированных файловых систем (`umount`). Для этой операции, которая тоже, как правило, является привилегией системного администратора, используется команда `umount` (может выполняться в ручном режиме или

автоматически при завершении работы операционной системы). Для пользователя обычно доступна команда отмонтирования файловой системы на дискете в форме

```
umount <имя точки монтирования>
```

где `<имя точки монтирования>` – это `<имя пустой директории>`, использованное ранее в команде `mount`, или в форме

```
umount /dev/fd0
```

где `/dev/fd0` – специальный файл устройства, соответствующего первому накопителю на гибких магнитных дисках.

Заметим, что для последующей корректной работы операционной системы при удалении физического носителя информации обязательно необходимо предварительное логическое разъединение файловых систем, если они перед этим были объединены.

Команда umount

Синтаксис команды

```
umount [-hV]
umount device
umount dir
```

Описание команды

Настоящее описание не является полным описанием команды `umount`, а описывает только те ее опции (очень малую часть), которые используются в данном курсе. Для получения полного описания следует обратиться к UNIX Manual (команда `man`).

Команда `umount` предназначена для выполнения операции логического разъединения ранее смонтированных файловых систем.

Опции `-h`, `-V` используются при вызове команды без параметров и служат для следующих целей:

`-h` – вывести краткую инструкцию по пользованию командой;

`-V` – вывести информацию о версии команды `umount`.

Команда `umount` с параметром служит для выполнения операции логического разъединения файловых систем. В качестве параметра может быть задано либо имя устройства, содержащего файловую систему – `device`, либо имя точки монтирования файловой системы (т. е. имя директории, которое указывалось в качестве параметра при вызове команды `mount`) – `dir`.

Заметим, что файловая система не может быть отмонтирована до тех пор, пока она находится в использовании (busy) – например, когда в ней существуют открытые файлы, какой-либо процесс имеет в качестве рабочей директории директорию в этой файловой системе и т. д.

Блочные, символьные устройства. Понятие драйвера. Блочные, символьные драйверы, драйверы низкого уровня. Файловый интерфейс

Обремененные знаниями об устройстве современных файловых систем в UNIX, мы можем, наконец, заняться вопросами реализации подсистемы ввода-вывода.

В лекции 13 (раздел «Структура системы ввода-вывода») речь шла о том, что все устройства ввода-вывода можно разделить на относительно небольшое число типов, в зависимости от набора операций, которые могут ими выполняться. Такое деление позволяет организовать «слоистую» структуру подсистемы ввода-вывода, вынеся все аппаратно-зависимые части в драйверы устройств, с которыми взаимодействует базовая подсистема ввода-вывода, осуществляющая стратегическое управление всеми устройствами.

В операционной системе UNIX принята упрощенная классификация устройств (см. лекцию 13, раздел «Систематизация внешних устройств и интерфейс между базовой подсистемой ввода-вывода и драйверами»): все устройства разделяются по способу передачи данных на символьные и блочные. Символьные устройства осуществляют передачу данных байт за байтом, в то время как блочные устройства передают блок байт как единое целое. Типичным примером символьного устройства является клавиатура, примером блочного устройства – жесткий диск. Непосредственное взаимодействие операционной системы с устройствами ввода-вывода обеспечивают их драйверы. Существует пять основных случаев, когда ядро обращается к драйверам.

1. Автоконфигурация. Происходит в процессе инициализации операционной системы, когда ядро определяет наличие доступных устройств.
2. Ввод-вывод. Обработка запроса ввода-вывода.
3. Обработка прерываний. Ядро вызывает специальные функции драйвера для обработки прерывания, поступившего от устройства, в том числе, возможно, для планирования очередности запросов к нему.
4. Специальные запросы. Например, изменение параметров драйвера или устройства.

5. Повторная инициализация устройства или останов операционной системы.

Так же как устройства подразделяются на символьные и блочные, драйверы тоже существуют символьные и блочные. Особенностью блочных устройств является возможность организации на них файловой системы, поэтому блочные драйверы обычно используются файловой системой UNIX. При обращении к блочному устройству, не содержащему файловой системы, применяются специальные драйверы низкого уровня, как правило, представляющие собой интерфейс между ядром операционной системы и блочным драйвером устройства.

Для каждого из этих трех типов драйверов были выделены основные функции, которые базовая подсистема ввода-вывода может совершать над устройствами и драйверами: инициализация устройства или драйвера, временное завершение работы устройства, чтение, запись, обработка прерывания, опрос устройства и т. д. (об этих операциях уже говорилось в лекции 13, раздел «Систематизация внешних устройств и интерфейс между базовой подсистемой ввода-вывода и драйверами»). Эти функции были систематизированы и представляют собой интерфейс между драйверами и базовой подсистемой ввода-вывода.

Каждый драйвер определенного типа в операционной системе UNIX получает собственный номер, который по сути дела является индексом в массиве специальных структур данных операционной системы – *коммутаторе устройств* соответствующего типа. Этот индекс принято также называть *старшим номером устройства*, хотя на самом деле он относится не к устройству, а к драйверу. Несмотря на наличие трех типов драйверов, в операционной системе используется всего два коммутатора: для блочных и символьных драйверов. Драйверы низкого уровня распределяются между ними по преобладающему типу интерфейса (к какому типу ближе – в такой массив и заносятся). Каждый элемент коммутатора устройств обязательно содержит адреса (точки входа в драйвер), соответствующие стандартному набору функций интерфейса, которые и вызываются операционной системой для выполнения тех или иных действий над устройством и/или драйвером.

Помимо старшего номера устройства существует еще и *младший номер устройства*, который передается драйверу в качестве параметра и смысл которого определяется самим драйвером. Например, это может быть номер раздела на жестком диске (partition), доступ к которому должен обеспечить драйвер (надо отметить, что в операционной системе UNIX различные разделы физического носителя информации рассматриваются как различные устройства). В некоторых случаях младший номер устройства может не использоваться, но для единообразия он должен присутствовать. Таким образом, пара драйвер–устройство

всегда однозначно определяется в операционной системе заданием пары номеров (старшего и младшего номеров устройства) и типа драйвера (символьный или блочный).

Для связи приложений с драйверами устройств операционная система UNIX использует *файловый интерфейс*. В числе типов файлов на предыдущем семинаре упоминались специальные файлы устройств. Так вот, каждой тройке тип–драйвер–устройство в файловой системе соответствует специальный файл устройства, который не занимает на диске никаких логических блоков, кроме индексного узла. В качестве атрибутов этого файла помимо обычных атрибутов используются соответствующие старший и младший номера устройства и тип драйвера (тип драйвера определяется по типу файла: ибо есть специальные файлы символьных устройств и специальные файлы блочных устройств, а номера устройств занимают место длины файла, скажем, для регулярных файлов). Когда открывается специальный файл устройства, операционная система, в числе прочих действий, заносит в соответствующий элемент таблицы открытых виртуальных узлов указатель на набор функций интерфейса из соответствующего элемента коммутатора устройств. Теперь при попытке чтения из файла устройства или записи в файл устройства виртуальная файловая система будет транслировать запросы на выполнение этих операций в соответствующие вызовы нужного драйвера.

Мы не будем останавливаться на практическом применении файлового интерфейса для работы с устройствами ввода-вывода, поскольку это выходит за пределы нашего курса, а вместо этого приступим к изложению концепции сигналов в UNIX, тесно связанных с понятиями аппаратного прерывания, исключения и программного прерывания.

Аппаратные прерывания (interrupt), исключения (exception), программные прерывания (trap, software interrupt). Их обработка

В лекции 13 (раздел «Опрос устройств и прерывания. Исключительные ситуации и системные вызовы») уже вводились понятия аппаратного прерывания, исключения и программного прерывания. Кратко напомним сказанное.

После выдачи запроса ввода-вывода у процессора существует два способа узнать о том, что обработка запроса устройством завершена. Первый способ заключается в регулярной проверке процессором бита занятости в регистре состояния контроллера соответствующего устройства (polling). Второй способ заключается в использовании прерываний. При втором способе процессор имеет специальный вход, на кото-

рый устройства ввода-вывода, используя контроллер прерываний или непосредственно, выставляют сигнал запроса прерывания (interrupt request) при завершении операции ввода-вывода. При наличии такого сигнала процессор после выполнения текущей команды не выполняет следующую, а, сохранив состояние ряда регистров и, возможно, загрузив в часть регистров новые значения, переходит к выполнению команд, расположенных по некоторым фиксированным адресам. После окончания обработки прерывания можно восстановить состояние процессора и продолжить его работу с команды, выполнение которой было отложено.

Аналогичный механизм часто используется при обработке исключительных ситуаций (exception), возникающих при выполнении команды процессором (неправильный адрес в команде, защита памяти, деление на ноль и т. д.). В этом случае процессор не завершает выполнение команды, а поступает, как и при прерывании, сохраняя свое состояние до момента начала ее выполнения.

Этим же механизмом часто пользуются и для реализации так называемых программных прерываний (software interrupt, trap), применяемых, например, для переключения процессора из режима пользователя в режим ядра внутри системных вызовов. Для выполнения действий, аналогичных действиям по обработке прерывания, процессор в этом случае должен выполнить специальную команду.

Необходимо четко представлять себе разницу между этими тремя понятиями, для чего не лишним будет в очередной раз обратиться к лекциям (лекция 13, раздел «Опрос устройств и прерывания. Исключительные ситуации и системные вызовы»).

Как правило, обработку аппаратных прерываний от устройств ввода-вывода производит сама операционная система, не доверяя работу с системными ресурсами процессам пользователя. Обработка же исключительных ситуаций и некоторых программных прерываний вполне может быть возложена на пользовательский процесс через механизм сигналов.

Понятие сигнала. Способы возникновения сигналов и виды их обработки

С точки зрения пользователя получение процессом сигнала выглядит как возникновение прерывания. Процесс прекращает регулярное исполнение, и управление передается механизму обработки сигнала. По окончании обработки сигнала процесс может возобновить регулярное исполнение. Типы сигналов (их принято задавать номерами, как правило, в диапазоне от 1 до 31 включительно или специальными символьными обозначениями) и способы их возникновения в системе жестко регламентированы.

Процесс может получить сигнал от:

- hardware (при возникновении исключительной ситуации);
- другого процесса, выполнившего системный вызов передачи сигнала;
- операционной системы (при наступлении некоторых событий);
- терминала (при нажатии определенной комбинации клавиш);
- системы управления заданиями (при выполнении команды kill – мы рассмотрим ее позже).

Передачу сигналов процессу в случаях его генерации источниками 2, 3 и 5, т. е., в конечном счете, каким-либо другим процессом, можно рассматривать как реализацию в UNIX сигнальных средств связи, о которых рассказывалось в лекции 4.

Существует три варианта реакции процесса на сигнал:

1. Принудительно проигнорировать сигнал.
2. Произвести обработку по умолчанию: проигнорировать, остановить процесс (перевести в состояние ожидания до получения другого специального сигнала), либо завершить работу с образованием core файла или без него.
3. Выполнить обработку сигнала, специфицированную пользователем.

Изменить реакцию процесса на сигнал можно с помощью специальных системных вызовов, которые мы рассмотрим позже. Реакция на некоторые сигналы не допускает изменения, и они могут быть обработаны только по умолчанию. Так, например, сигнал с номером 9 – SIGKILL обрабатывается только по умолчанию и всегда приводит к завершению процесса.

Важным вопросом при программировании с использованием сигналов является вопрос о сохранении реакции на них при порождении нового процесса или замене его пользовательского контекста. При системном вызове fork() все установленные реакции на сигналы наследуются порожденным процессом.

При системном вызове exec() **сохраняются реакции только для тех сигналов, которые игнорировались или обрабатывались по умолчанию. Получение любого сигнала, который до вызова exec() обрабатывался пользователем, приведет к завершению процесса.**

Прежде чем продолжить тему сигналов, нам придется подробнее остановиться на иерархии процессов в операционной системе.

Понятия группы процессов, сеанса, лидера группы, лидера сеанса, управляющего терминала сеанса. Системные вызовы getpgrp(), setpgrp(), getpgid(), setpgid(), getsid(), setsid()

В лекции 2, раздел «Одноразовые операции», уже говорилось, что все процессы в системе связаны родственными отношениями и образуют генеалогическое дерево или лес из таких деревьев, где в качестве узлов деревьев выступают сами процессы, а связями служат отношения родитель-ребенок. Все эти деревья принято разделять на *группы процессов*, или семьи (см. рис. 12-13.1).

Группа процессов включает в себя один или более процессов и существует, пока в группе присутствует хотя бы один процесс. Каждый процесс обязательно включен в какую-нибудь группу. При рождении нового процесса он попадает в ту же группу процессов, в которой находится его родитель. Процессы могут мигрировать из группы в группу по своему желанию или по желанию другого процесса (в зависимости от версии UNIX). Многие системные вызовы могут быть применены не к одному конкретному процессу, а ко всем процессам в некоторой группе. Поэтому то, как именно следует объединять процессы в группы, зависит от того, как предполагается их использовать. Чуть позже мы поговорим об использовании групп процессов для передачи сигналов.

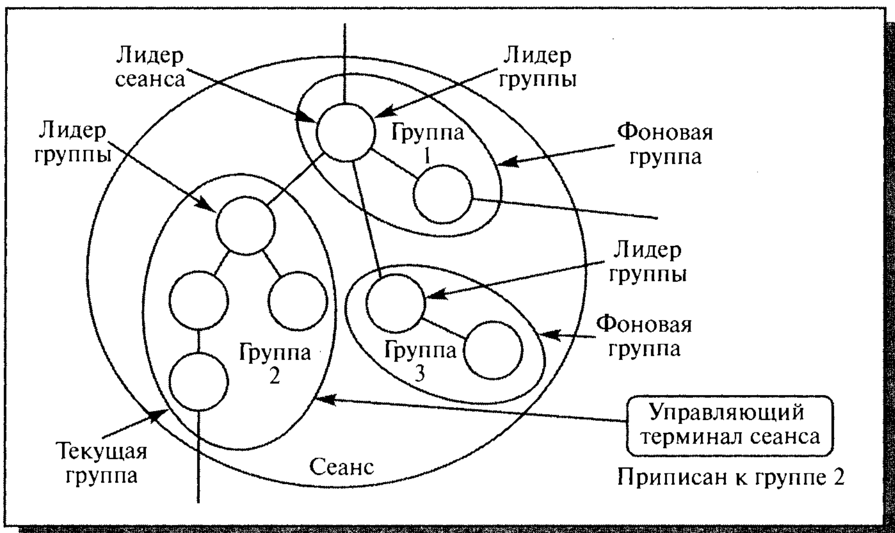


Рис. 12-13.1. Иерархия процессов в UNIX

В свою очередь, группы процессов объединяются в *сеансы*, образуя, с родственной точки зрения, некие кланы семей. Понятие сеанса изначально было введено в UNIX для логического объединения групп процессов, созданных в результате каждого входа и последующей работы пользователя в системе. С каждым сеансом, поэтому, может быть связан в системе терминал, называемый *управляющим терминалом сеанса*, через который обычно и общаются процессы сеанса с пользователем. Сеанс не может иметь более одного управляющего терминала, и один терминал не может быть управляющим для нескольких сеансов. В то же время могут существовать сеансы, вообще не имеющие управляющего терминала.

Каждая группа процессов в системе получает собственный уникальный номер. Узнать этот номер можно с помощью системного вызова `getpgid()`. Используя его, процесс может узнать номер группы для себя самого или для процесса из своего сеанса. К сожалению, не во всех версиях UNIX присутствует данный системный вызов. Здесь мы сталкиваемся с тяжелым наследием разделения линий UNIX'ов на линию BSD и линию System V, которое будет нас преследовать почти на всем протяжении данной темы. Вместо вызова `getpgid()` в таких системах существует системный вызов `getpgrp()`, который возвращает номер группы только для текущего процесса.

Системный вызов getpgid()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgid(pid_t pid);
```

Описание системного вызова

Системный вызов `getpgid` возвращает идентификатор группы процессов для процесса с идентификатором `pid`.

Узнать номер группы процесс может только для себя самого или для процесса из своего сеанса. При других значениях `pid` системный вызов возвращает значение `-1`.

Тип данных `pid_t` является синонимом для одного из целочисленных типов языка С.

Системный вызов getpgrp()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgrp(void);
```

Описание системного вызова

Системный вызов getpgrp возвращает идентификатор группы процессов для текущего процесса.

Тип данных pid_t является синонимом для одного из целочисленных типов языка C.

Для перевода процесса в другую группу процессов (возможно, с одновременным ее созданием) применяется системный вызов setpgid(). Перевести в другую группу процесс может либо самого себя (и то не во всякую и не всегда), либо свой процесс-ребенок, который не выполнял системный вызов exec(), т. е. не запускал на выполнение другую программу. При определенных значениях параметров системного вызова создается новая группа процессов с идентификатором, совпадающим с идентификатором переводимого процесса, состоящая первоначально только из одного этого процесса. Новая группа может быть создана только таким способом, поэтому идентификаторы групп в системе уникальны. Переход в другую группу без создания новой группы возможен лишь в пределах одного сеанса.

В некоторых разновидностях UNIX системный вызов setpgid() отсутствует, а вместо него используется системный вызов setpgrp(), способный только создавать новую группу процессов с идентификатором, совпадающим с идентификатором текущего процесса, и переводить в нее текущий процесс. (В ряде систем, где сосуществуют вызовы setpgrp() и setpgid(), например в Solaris, вызов setpgrp() ведет себя иначе – он аналогичен рассматриваемому ниже вызову setsid().)

Системный вызов setpgid()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```

Описание системного вызова

Системный вызов setpgid служит для перевода процесса из одной группы процессов в другую, а также для создания новой группы процессов.

Параметр pid является идентификатором процесса, который нужно перевести в другую группу, а параметр pgid – идентификатором группы процессов, в которую предстоит перевести этот процесс.

Не все комбинации этих параметров разрешены. Перевести в другую группу процесс может либо самого себя (и то не во всякую, и не всегда), либо свой процесс-ребенок, который не выполнял системный вызов exec(), т. е. не запускал на выполнение другую программу.

- Если параметр pid равен 0, то считается, что процесс переводит в другую группу самого себя.
- Если параметр pgid равен 0, то в Linux считается, что процесс переводится в группу с идентификатором, совпадающим с идентификатором процесса, определяемого первым параметром.
- Если значения, определяемые параметрами pid и pgid, равны, то создается новая группа с идентификатором, совпадающим с идентификатором переводимого процесса, состоящая первоначально только из этого процесса. **Переход в другую группу без создания новой группы возможен только в пределах одного сеанса.**

В новую группу не может перейти процесс, являющийся лидером группы, т.е. процесс, идентификатор которого совпадает с идентификатором его группы.

Тип данных pid_t является синонимом для одного из целочисленных типов языка C.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Системный вызов setpgrp()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
int setpgrp(void);
```

Описание системного вызова

Системный вызов setpgrp служит для перевода текущего процесса во вновь создаваемую группу процессов, идентификатор которой будет совпадать с идентификатором текущего процесса.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Процесс, идентификатор которого совпадает с идентификатором его группы, называется *лидером группы*. Одно из ограничений на применение вызовов `setpgid()` и `setpgrp()` состоит в том, что лидер группы не может перебраться в другую группу.

Каждый сеанс в системе также имеет собственный номер. Для того чтобы узнать его, можно воспользоваться системным вызовом `getsid()`. В разных версиях UNIX на него накладываются различные ограничения. В Linux такие ограничения отсутствуют.

Системный вызов getsid()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
pid_t getsid(pid_t pid);
```

Описание системного вызова

Системный вызов `getpgid` возвращает идентификатор сеанса для процесса с идентификатором `pid`. Если параметр `pid` равен `0`, то возвращается идентификатор сеанса для данного процесса.

Тип данных `pid_t` является синонимом для одного из целочисленных типов языка C.

Использование системного вызова `setsid()` приводит к созданию новой группы, состоящей только из процесса, который его выполнил (он становится лидером новой группы), и нового сеанса, идентификатор которого совпадает с идентификатором процесса, сделавшего вызов. Такой процесс называется *лидером сеанса*. Этот системный вызов может применять только процесс, не являющийся лидером группы.

Системный вызов setsid()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
int setsid(void);
```

Описание системного вызова

Этот системный вызов может применять только процесс, не являющийся лидером группы, т. е. процесс, идентификатор которого не совпадает с идентификатором его группы. Использование системного вызова `setsid` приводит к созданию новой группы, состоящей только из

процесса, который его выполнил (он становится лидером новой группы), и нового сеанса, идентификатор которого совпадает с идентификатором процесса, сделавшего вызов.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Если сеанс имеет управляющий терминал, то этот терминал обязательно приписывается к некоторой группе процессов, входящей в сеанс. Такая группа процессов *называется текущей группой процессов* для данного сеанса. Все процессы, входящие в текущую группу процессов, могут совершать операции ввода-вывода, используя управляющий терминал. Все остальные группы процессов сеанса называются *фоновыми группами*, а процессы, входящие в них – *фоновыми процессами*. При попытке ввода-вывода фонового процесса через управляющий терминал этот процесс получит сигналы, которые стандартно приводят к прекращению работы процесса. Передавать управляющий терминал от одной группы процессов к другой может только лидер сеанса. Заметим, что для сеансов, не имеющих управляющего терминала, все процессы являются фоновыми.

При завершении работы процесса – лидера сеанса все процессы из текущей группы сеанса получают сигнал SIGHUP, который при стандартной обработке приведет к их завершению. Таким образом, после завершения лидера сеанса в нормальной ситуации работу продолжат только фоновые процессы.

Процессы, входящие в текущую группу сеанса, могут получать сигналы, инициируемые нажатием определенных клавиш на терминале – SIGINT при нажатии клавиш <CTRL> и <C>, и SIGQUIT при нажатии клавиш <CTRL> и <4>. Стандартная реакция на эти сигналы – завершение процесса (с образованием core файла для сигнала SIGQUIT).

Необходимо ввести еще одно понятие, связанное с процессом, – эффективный идентификатор пользователя. В материалах первого семинара говорилось о том, что каждый пользователь в системе имеет собственный идентификатор – UID. Каждый процесс, запущенный пользователем, задействует этот UID для определения своих полномочий. Однако иногда, если у исполняемого файла были выставлены соответствующие атрибуты, процесс может выдать себя за процесс, запущенный другим пользователем. Идентификатор пользователя, от имени которого процесс пользуется полномочиями, и является эффективным идентификатором пользователя для процесса – EUID. За исключением выше оговоренного случая, эффективный идентификатор пользователя совпадает с идентификатором пользователя, создавшего процесс.

Системный вызов kill() и команда kill()

Из всех перечисленных ранее в разделе «Аппаратные прерывания (interrupt), исключения (exception), программные прерывания (trap, software interrupt). Их обработка» источников сигнала пользователю доступны только два – команда `kill` и посылка сигнала процессу с помощью системного вызова `kill()`. Команда `kill` обычно используется в следующей форме:

```
kill [-номер] pid
```

Здесь `pid` – это идентификатор процесса, которому посылается сигнал, а `номер` – номер сигнала, который посылается процессу. Послать сигнал (если у вас нет полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с идентификатором пользователя, посылающего сигнал. Если параметр `-номер` отсутствует, то посылается сигнал `SIGTERM`, обычно имеющий номер `15`, и реакция на него по умолчанию – завершить работу процесса, который получил сигнал.

Команда kill

Синтаксис команды

```
kill [-signal] [--] pid
kill -l
```

Описание команды

Команда `kill` предназначена для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя.

Параметр `pid` определяет процесс или процессы, которым будут доставляться сигналы. Он может быть задан одним из следующих четырех способов:

- Число `n > 0` – определяет идентификатор процесса, которому будет доставлен сигнал.
- Число `0` – сигнал будет доставлен всем процессам текущей группы для данного управляющего терминала.
- Число `-1` с предваряющей опцией `'--'` – сигнал будет доставлен (если позволяют полномочия) всем процессам с идентификаторами, большими `1`.
- Число `n < 0`, где `n` не равно `-1`, с предваряющей опцией `'--'` – сигнал будет доставлен всем процессам из группы процессов, идентификатор которой равен `-n`.

Параметр `-signal` определяет тип сигнала, который должен быть доставлен, и может задаваться в числовой или символьной форме, например `-9` или `-SIGKILL`. Если этот параметр опущен, процессам по умолчанию посылается сигнал `SIGTERM`.

Послать сигнал (не имея полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с идентификатором пользователя, посылающего сигнал.

Опция -l используется для получения списка сигналов, существующих в системе в символьной и числовой формах.

Во многих операционных системах предусмотрены еще и дополнительные опции для команды kill.

При использовании системного вызова kill() послать сигнал (не имея полномочий суперпользователя) можно только процессу или процессам, у которых эффективный идентификатор пользователя совпадает с эффективным идентификатором пользователя процесса, посылающего сигнал.

Системный вызов kill()

Прототип системного вызова

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signal);
```

Описание системного вызова

Системный вызов kill() предназначен для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя.

Послать сигнал (не имея полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с эффективным идентификатором пользователя для процесса, посылающего сигнал.

Аргумент pid описывает, кому посылается сигнал, а аргумент sig – какой сигнал посылается. Этот системный вызов умеет делать много разных вещей, в зависимости от значения аргументов:

• Если pid > 0 и sig > 0, то сигнал номером sig (если позволяют привилегии) посылается процессу с идентификатором pid.

• Если pid = 0, а sig > 0, то сигнал с номером sig посылается всем процессам в группе, к которой принадлежит посылающий процесс.

• Если pid = -1, sig > 0 и посылающий процесс не является процессом суперпользователя, то сигнал посылается всем процессам в системе, для которых идентификатор пользователя совпадает с эффективным идентификатором пользователя процесса, посылающего сигнал.

• Если pid = -1, sig > 0 и посылающий процесс является процессом суперпользователя, то сигнал посылается всем процессам в системе, за исключением системных процессов (обычно всем, кроме процессов с pid = 0 и pid = 1).

• Если pid < 0, но не -1, sig > 0, то сигнал посылается всем процессам из группы, идентификатор которой равен абсолютному значению аргумента pid (если позволяют привилегии).

• Если значение sig = 0, то производится проверка на ошибку, а сигнал не посылается, так как все сигналы имеют номера > 0. Это можно использовать для проверки правильности аргумента pid (есть ли в системе процесс или группа процессов с соответствующим идентификатором).

Возвращаемое значение

Системный вызов возвращает 0 при нормальном завершении и -1 при ошибке.

Изучение особенностей получения терминальных сигналов текущей и фоновой группой процессов

Возьмем тривиальную программу 12–13-1.c, в которой процесс порождает ребенка, и они оба зацикливаются, и на ее основе проиллюстрируем сказанное выше:

```
/* Тривиальная программа для иллюстрации понятий
группа процессов, сеанс, фоновая группа и т. д. */
#include <unistd.h>
int main(void){
   (void)fork();
   while(1);
   return 0;
}
```

Для этого будем использовать команду ps с опциями -e и j, которая позволяет получить информацию обо всех процессах в системе и узнать их идентификаторы, идентификаторы групп процессов и сеансов, управляющий терминал сеанса и к какой группе процессов он приписан. Набрав команду "ps -e j" **(обратите внимание на наличие пробела между буквами e и j!!!)** мы получим список всех процессов в системе. Колонка PID содержит идентификаторы процессов, колонка PGID – идентификаторы групп, к которым они принадлежат, колонка SID – идентификаторы сеансов, колонка TTY – номер соответствующего управляющего терминала, колонка TPGID (может присутствовать не во всех версиях UNIX, но в Linux она есть) – к какой группе процессов приписан управляющий терминал.

Наберите тривиальную программу, откомпилируйте ее и запустите на исполнение (лучше всего из-под оболочки Midnight Commander – mc). Запустив команду "ps -e j" с другого экрана, проанализируйте значения

идентификаторов группы процессов, сеансов, прикрепления управляющего терминала, текущей и фоновой групп. Убедитесь, что тривиальные процессы относятся к текущей группе сеанса. Проверьте реакцию текущей группы на сигналы SIGINT – нажатие клавиш <CTRL> и <C> – и SIGQUIT – нажатие клавиш <CTRL> и <4>.

Запустите теперь тривиальную программу в фоновом режиме, например командой "a.out &". Проанализируйте значения идентификаторов группы процессов, сеансов, прикрепления управляющего терминала, текущей и фоновой групп. Убедитесь, что тривиальные процессы относятся к фоновой группе сеанса. Проверьте реакцию фоновой группы на сигналы SIGINT – нажатие клавиш <CTRL> и <C>, и SIGQUIT – нажатие клавиш <CTRL> и <4>. Ликвидируйте тривиальные процессы с помощью команды kill.

Изучение получения сигнала SIGHUP процессами при завершении лидера сеанса

Возьмите снова тривиальную программу из предыдущего раздела и запустите ее на исполнение из-под Midnight Commander в текущей группе. Проанализировав значения идентификаторов группы процессов, сеансов, прикрепления управляющего терминала, текущей и фоновой групп, ликвидируйте лидера сеанса для тривиальных процессов. Убедитесь, что все процессы в текущей группе этого сеанса прекратили свою работу.

Запустите тривиальную программу в фоновом режиме. Снова удалите лидера сеанса для тривиальных процессов. Убедитесь, что фоновая группа продолжает работать. Ликвидируйте тривиальные процессы.

Системный вызов signal(). Установка собственного обработчика сигнала

Одним из способов изменения поведения процесса при получении сигнала в операционной системе UNIX является использование системного вызова signal().

Системный вызов signal()

Прототип системного вызова

```
#include <signal.h>
void (*signal (int sig,
   void (*handler) (int)))(int);
```

Описание системного вызова

Системный вызов signal служит для изменения реакции процесса на какой-либо сигнал. Хотя прототип системного вызова выглядит довольно пугающе, ничего страшного в нем нет. Приведенное выше описание можно словесно изложить следующим образом:

> функция signal, возвращающая указатель на функцию с одним параметром типа int, которая ничего не возвращает, и имеющая два параметра: параметр sig типа int и параметр handler, служащий указателем на ничего не возвращающую функцию с одним параметром типа int.

Параметр sig – это номер сигнала, обработку которого предстоит изменить.

Параметр handler описывает новый способ обработки сигнала – это может быть указатель на пользовательскую функцию – обработчик сигнала, специальное значение SIG_DFL или специальное значение SIG_IGN. Специальное значение SIG_IGN используется для того, чтобы процесс игнорировал поступившие сигналы с номером sig, специальное значение SIG_DFL – для восстановления реакции процесса на этот сигнал по умолчанию.

Возвращаемое значение

Системный вызов возвращает указатель на старый способ обработки сигнала, значение которого можно использовать для восстановления старого способа в случае необходимости.

Этот системный вызов имеет два параметра: один из них задает номер сигнала, реакцию процесса на который требуется изменить, а второй определяет, как именно мы собираемся ее менять. Для первого варианта реакции процесса на сигнал (см. раздел «Понятие сигнала. Способы возникновения сигналов и виды их обработки») – его игнорирования – применяется специальное значение этого параметра — SIG_IGN. Например, если требуется игнорировать сигнал SIGINT, начиная с некоторого места работы программы, в этом месте программы мы должны употребить конструкцию

```
(void) signal(SIGINT, SIG_IGN);
```

Для второго варианта реакции процесса на сигнал – восстановления его обработки по умолчанию – применяется специальное значение этого параметра — SIG_DFL. Для третьего варианта реакции процесса на сигнал в качестве значения параметра подставляется указатель на пользовательскую функцию обработки сигнала, которая должна иметь прототип вида

```
void *handler(int);
```

Ниже приведен пример скелета конструкции для пользовательской обработки сигнала SIGHUP:

```
void *my_handler(int nsig) {
  <обработка сигнала>
}
int main() {
  ...
  (void)signal(SIGHUP, my_handler);
  ...
}
```

В качестве значения параметра в пользовательскую функцию обработки сигнала (в нашем скелете – параметр nsig) передается номер возникшего сигнала, так что одна и та же функция может быть использована для обработки нескольких сигналов.

Прогон программы, игнорирующей сигнал SIGINT

Рассмотрим следующую программу – 12–13-2.c:

```
/* Программа, игнорирующая сигнал SIGINT */
#include <signal.h>
int main(void){
  /* Выставляем реакцию процесса на сигнал SIGINT
  на игнорирование */
  (void)signal(SIGINT, SIG_IGN);
  /*Начиная с этого места, процесс будет игнорировать
  возникновение сигнала SIGINT */
  while(1);
  return 0;
}
```

Эта программа не делает ничего полезного, кроме переустановки реакции на нажатие клавиш <CTRL> и <C> на игнорирование возникающего сигнала и своего бесконечного зацикливания. Наберите, откомпилируйте и запустите эту программу, убедитесь, что на нажатие клавиш <CTRL> и <C> она не реагирует, а реакция на нажатие клавиш <CTRL> и <4> осталась прежней.

Модификация предыдущей программы для игнорирования сигналов SIGINT и SIGQUIT

Модифицируйте программу из предыдущего раздела так, чтобы она перестала реагировать и на нажатие клавиш <CTRL> и <4>. Откомпилируйте и запустите ее, убедитесь в отсутствии ее реакций на внешние раздражители. Снимать программу придется теперь с другого терминала командой `kill`.

Прогон программы с пользовательской обработкой сигнала SIGINT

Рассмотрим теперь другую программу – 12–13-3.с:

```
/* Программа с пользовательской обработкой сигнала
SIGINT */
#include <signal.h>
#include <stdio.h>
/* Функция my_handler - пользовательский обработчик
сигнала */
void my_handler(int nsig){
  printf("Receive signal %d, CTRL-C pressed\n", nsig);
}
int main(void){
  /* Выставляем реакцию процесса на сигнал SIGINT */
  (void)signal(SIGINT, my_handler);
  /*Начиная с этого места, процесс будет печатать
  сообщение о возникновении сигнала SIGINT */
  while(1);
  return 0;
}
```

Эта программа отличается от программы из раздела «Прогон программы, игнорирующей сигнал SIGINT» тем, что в ней введена обработка сигнала `SIGINT` пользовательской функцией. Наберите, откомпилируйте и запустите эту программу, проверьте ее реакцию на нажатие клавиш <CTRL> и <C> и на нажатие клавиш <CTRL> и <4>.

Модификация предыдущей программы для пользовательской обработки сигналов SIGINT и SIGQUIT

Модифицируйте программу из предыдущего раздела так, чтобы она печатала сообщение и о нажатии клавиш <CTRL> и <4>. Используйте одну и ту же функцию для обработки сигналов SIGINT и SIGQUIT. Откомпилируйте и запустите ее, проверьте корректность работы. Снимать программу также придется с другого терминала командой kill.

Восстановление предыдущей реакции на сигнал

До сих пор в примерах мы игнорировали значение, возвращаемое системным вызовом signal(). На самом деле этот системный вызов возвращает указатель на предыдущий обработчик сигнала, что позволяет восстанавливать переопределенную реакцию на сигнал. Рассмотрим пример программы 12—13-4.c, возвращающей первоначальную реакцию на сигнал SIGINT после 5 пользовательских обработок сигнала:

```
/* Программа с пользовательской обработкой сигнала
SIGINT, возвращающаяся к первоначальной реакции на
этот сигнал после 5 его обработок*/
#include <signal.h>
#include <stdio.h>
int i=0; /* Счетчик числа обработок сигнала */
void (*p)(int); /* Указатель, в который будет
занесен адрес предыдущего обработчика сигнала */
/* Функция my_handler - пользовательский обработчик
  сигнала */
void my_handler(int nsig){
  printf("Receive signal %d, CTRL-C pressed\n", nsig);
  i = i+1;
  /* После 5-й обработки возвращаем первоначальную
  реакцию на сигнал */
  if(i == 5) (void)signal(SIGINT, p);
}
int main(void){
  /* Выставляем свою реакцию процесса на сигнал
  SIGINT, запоминая адрес предыдущего обработчика */
  p = signal(SIGINT, my_handler);
```

```
    /*Начиная с этого места, процесс будет 5 раз
    печатать сообщение о возникновении сигнала SIGINT */
    while(1);
    return 0;
}
```

Наберите, откомпилируйте программу и запустите ее на исполнение.

Сигналы SIGUSR1 и SIGUSR2. Использование сигналов для синхронизации процессов

В операционной системе UNIX существует два сигнала, источниками которых могут служить только системный вызов `kill()` или команда `kill` – это сигналы `SIGUSR1` и `SIGUSR2`. Обычно их применяют для передачи информации о происшедшем событии от одного пользовательского процесса другому в качестве сигнального средства связи.

В материалах семинара 5 (раздел «Написание, компиляция и запуск программы для организации двунаправленной связи между родственными процессами через pipe»), когда рассматривалась связь родственных процессов через pipe, речь шла о том, что pipe является однонаправленным каналом связи, и что для организации связи через один pipe в двух направлениях необходимо задействовать механизмы взаимной синхронизации процессов. Организуйте двустороннюю поочередную связь процесса-родителя и процесса-ребенка через pipe, используя для синхронизации сигналы `SIGUSR1` и `SIGUSR2`, модифицировав программу из раздела. «Прогон программы для организации однонаправленной связи между родственными процессами через pipe» семинара 5.

Задача повышенной сложности: организуйте побитовую передачу целого числа между двумя процессами, используя для этого только сигналы `SIGUSR1` и `SIGUSR2`.

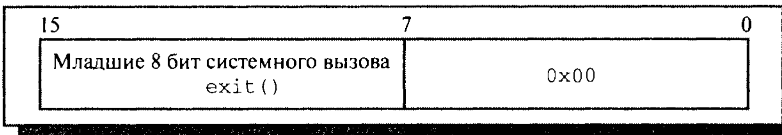
При реализации нитей исполнения в операционной системе Linux (см. семинары 6–7, начиная с раздела «Понятие о нити исполнения (thread) в UNIX. Идентификатор нити исполнения. Функция pthread_self()») сигналы `SIGUSR1` и `SIGUSR2` используются для организации синхронизации между процессами, представляющими нити исполнения, и процессом-координатором в служебных целях. Поэтому пользовательские программы, применяющие в своей работе нити исполнения, **не могут задействовать сигналы** `SIGUSR1` и `SIGUSR2`.

Завершение порожденного процесса. Системный вызов waitpid(). Сигнал SIGCHLD

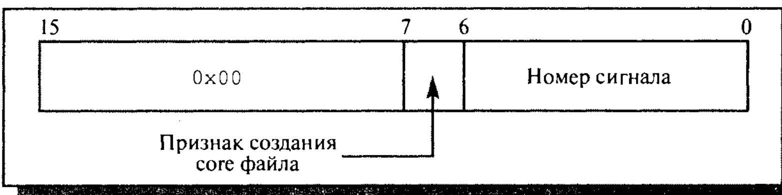
В материалах семинаров 3–4 (раздел «Завершение процесса. Функция exit()») при изучении завершения процесса говорилось о том, что если процесс-ребенок завершает свою работу прежде процесса-родителя, и процесс-родитель явно не указал, что он не заинтересован в получении информации о статусе завершения процесса-ребенка, то завершившийся процесс не исчезает из системы окончательно, а остается в состоянии ***закончил исполнение*** (зомби-процесс) либо до завершения процесса-родителя, либо до того момента, когда родитель соблаговолит получить эту информацию.

Для получения такой информации процесс-родитель может воспользоваться системным вызовом `waitpid()` или его упрощенной формой `wait()`. Системный вызов `waitpid()` позволяет процессу-родителю синхронно получить данные о статусе завершившегося процесса-ребенка либо блокируя процесс-родитель до завершения процесса-ребенка, либо без блокировки при его периодическом вызове с опцией `WNOHANG`. Эти данные занимают 16 бит и в рамках нашего курса могут быть расшифрованы следующим образом:

- Если процесс завершился при помощи явного или неявного вызова функции `exit()`, то данные выглядят так (старший бит находится слева)



- Если процесс был завершен сигналом, то данные выглядят так (старший бит находится слева)



Каждый процесс-ребенок при завершении работы посылает своему процессу-родителю специальный сигнал `SIGCHLD`, на который у всех процессов по умолчанию установлена реакция «игнорировать сигнал». Наличие такого сигнала совместно с системным вызовом `waitpid()` позволяет организовать асинхронный сбор информации о статусе завершившихся порожденных процессов процессом-родителем.

Системные вызовы wait() и waitpid()

Прототипы системных вызовов

```
#include <sys/types.h>
#include <wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
pid_t wait(int *status);
```

Описание системных вызовов

Это описание не является полным описанием системных вызовов, а адаптировано применительно к нашему курсу. Для получения полного описания обращайтесь к UNIX Manual.

Системный вызов `waitpid()` блокирует выполнение текущего процесса до тех пор, пока либо не завершится порожденный им процесс, определяемый значением параметра `pid`, либо текущий процесс не получит сигнал, для которого установлена реакция по умолчанию «завершить процесс» или реакция обработки пользовательской функцией. Если порожденный процесс, заданный параметром `pid`, к моменту системного вызова находится в состоянии Œ Ł° Łæ ° - Ł , то системный вызов возвращается немедленно без блокирования текущего процесса.

Параметр `pid` определяет порожденный процесс, завершения которого дожидается процесс-родитель, следующим образом:

- Если `pid > 0` ожидаем завершения процесса с идентификатором `pid`.
- Если `pid = 0`, то ожидаем завершения любого порожденного процесса в группе, к которой принадлежит процесс-родитель.
- Если `pid = -1`, то ожидаем завершения любого порожденного процесса.
- Если `pid < 0`, но не `-1`, то ожидаем завершения любого порожденного процесса из группы, идентификатор которой равен абсолютному значению параметра `pid`.

Параметр `options` в нашем курсе может принимать два значения: `0` и `WNOHANG`. Значение `WNOHANG` требует немедленного возврата из вызова без блокировки текущего процесса в любом случае.

Если системный вызов обнаружил завершившийся порожденный процесс из числа специфицированных параметром `pid`, то этот процесс удаляется из вычислительной системы, а по адресу, указанному в параметре `status`, сохраняется информация о статусе его завершения. Параметр `status` может быть задан равным `NULL`, если эта информация не имеет для нас значения.

При обнаружении завершившегося процесса системный вызов возвращает его идентификатор. Если вызов был сделан с установленной опцией `WNOHANG`, и порожденный процесс, специфицированный параметром `pid`, существует, но еще не завершился, системный вызов вернет значение `0`. Во всех остальных случаях он возвращает отрицательное значение. Возврат из вызова, связанный с возникновением обработанного пользователем сигнала, может быть в этом случае идентифицирован по значению системной переменной `errno == EINTR`, и вызов может быть сделан снова.

Системный вызов `wait` является синонимом для системного вызова `waitpid` со значениями параметров `pid = -1`, `options = 0`.

Используя системный вызов signal(), мы можем явно установить игнорирование этого сигнала (SIG_IGN), тем самым проинформировав систему, что нас не интересует, каким образом завершатся порожденные процессы. В этом случае зомби-процессов возникать не будет, но и применение системных вызовов wait() и waitpid() будет запрещено.

Прогон программы для иллюстрации обработки сигнала SIGCHLD

Для закрепления материала рассмотрим пример программы 12—13-5.c с асинхронным получением информации о статусе завершения порожденного процесса:

```
/* Программа с асинхронным получением информации о
статусе двух завершившихся порожденных процессов */
#include <sys/types.h>
#include <unistd.h>
#include <waith>
#include <signal.h>
#include <stdio.h>
/* Функция my_handler - обработчик сигнала SIGCHLD */
void my_handler(int nsig){
  int status;
  pid_t pid;
  /* Опрашиваем статус завершившегося процесса и
  одновременно узнаем его идентификатор */
  if((pid = waitpid(-1, &status, 0)) < 0){
    /* Если возникла ошибка - сообщаем о ней и
    продолжаем работу */
    printf("Some error on waitpid errno = %d\n",
      errno);
  } else {
    /* Иначе анализируем статус завершившегося
    процесса */
    if ((status & 0xff) == 0) {
      /* Процесс завершился с явным или неявным
      вызовом функции exit() */
      printf("Process %d was exited with status %d\n",
      pid, status >> 8);
    } else if ((status & 0xff00) == 0){
      /* Процесс был завершен с помощью сигнала */
```

```
            printf("Process %d killed by signal %d %s\n",
            pid, status &0x7f,(status & 0x80) ?
            "with core file" : "without core file");
        }
    }
}
int main(void){
    pid_t pid;
    /* Устанавливаем обработчик для сигнала SIGCHLD */
    (void) signal(SIGCHLD, my_handler);
    /* Порождаем Child 1 */
    if((pid = fork()) < 0){
        printf("Can\'t fork child 1\n");
        exit(1);
    } else if (pid == 0){
        /* Child 1 - завершается с кодом 200 */
        exit(200);
    }
    /* Продолжение процесса-родителя - порождаем Child 2 */
    if((pid = fork()) < 0){
        printf("Can\'t fork child 2\n");
        exit(1);
    } else if (pid == 0){
        /* Child 2 - циклится, необходимо удалять с
        помощью сигнала! */
        while(1);
    }
    /* Продолжение процесса-родителя - уходим в цикл */
    while(1);
    return 0;
}
```

В этой программе родитель порождает два процесса. Один из них завершается с кодом 200, а второй зацикливается. Перед порождением процессов родитель устанавливает обработчик прерывания для сигнала `SIGCHLD`, а после их порождения уходит в бесконечный цикл. В обработчике прерывания вызывается `waitpid()` для любого порожденного процесса. Так как в обработчик мы попадаем, когда какой-либо из процессов завершился, системный вызов не блокируется, и мы можем получить информацию об идентификаторе завершившегося процесса и причине его завершения. Откомпилируйте программу и запустите ее на исполнение. Второй порожденный процесс завершайте с помощью команды `kill` с

каким-либо номером сигнала. Родительский процесс также будет необходимо завершать командой `kill`.

Возникновение сигнала SIGPIPE при попытке записи в pipe или FIFO, который никто не собирается читать

В материалах семинара 5 (раздел «Особенности поведения вызовов read() и write() для pip'а») при обсуждении работы с pip'ами и FIFO мы говорили, что для них системные вызовы `read()` и `write()` имеют определенные особенности поведения. Одной из таких особенностей является получение сигнала `SIGPIPE` процессом, который пытается записывать информацию в pipe или в FIFO в том случае, когда читать ее уже некому (нет ни одного процесса, который держит соответствующий pipe или FIFO открытым для чтения). **Реакция по умолчанию на этот сигнал – прекратить работу процесса.** Теперь мы уже можем написать корректную обработку этого сигнала пользователем, например, для элегантного прекращения работы пишущего процесса. Однако для полноты картины необходимо познакомиться с особенностями поведения некоторых системных вызовов при получении процессом сигналов во время их выполнения.

По ходу нашего курса мы представили читателям ряд системных вызовов, которые могут во время выполнения блокировать процесс. К их числу относятся системный вызов `open()` при открытии FIFO, системные вызовы `read()` и `write()` при работе с pip'ами и FIFO, системные вызовы `msgsnd()` и `msgrcv()` при работе с очередями сообщений, системный вызов `semop()` при работе с семафорами и т. д. Что произойдет с процессом, если он, выполняя один из этих системных вызовов, получит какой-либо сигнал? Дальнейшее поведение процесса зависит от установленной для него реакции на этот сигнал:

- Если реакция на полученный сигнал была «игнорировать сигнал» (независимо от того, установлена она по умолчанию или пользователем с помощью системного вызова `signal()`), то поведение процесса не изменится.
- Если реакция на полученный сигнал установлена по умолчанию и заключается в прекращении работы процесса, то процесс перейдет в состояние ***закончил исполнение***.
- Если реакция процесса на сигнал заключается в выполнении пользовательской функции, то процесс выполнит эту функцию (если он находился в состоянии ***ожидание***, он попадет в состояние ***готовность*** и затем в состояние ***исполнение***) и вернется из системного вызова с констатацией ошибочной ситуации (некоторые системные вызовы позволяют операционной системе после выполнения обработки сигнала вновь вернуть процесс в состояние ожидания). Отличить

такой возврат от действительно ошибочной ситуации можно с помощью значения системной переменной errno, которая в этом случае примет значение EINTR (для вызова write и сигнала SIGPIPE соответствующее значение в порядке исключения будет EPIPE).

После этого краткого обсуждения становится до конца ясно, как корректно обработать ситуацию «никто не хотел прочитать» для системного вызова write(). Чтобы пришедший сигнал SIGPIPE не завершил работу нашего процесса по умолчанию, мы должны его обработать самостоятельно (функция-обработчик при этом может быть и пустой!). Но этого мало. Поскольку нормальный ход выполнения системного вызова был нарушен сигналом, мы вернемся из него с отрицательным значением, которое свидетельствует об ошибке. Проанализировав значение системной переменной errno на предмет совпадения со значением EPIPE, мы можем отличить возникновение сигнала SIGPIPE от других ошибочных ситуаций (неправильные значения параметров и т. д.) и грациозно продолжить работу программы.

Понятие о надежности сигналов. POSIX-функции для работы с сигналами

Основным недостатком системного вызова signal() является его низкая надежность.

Во многих вариантах операционной системы UNIX установленная при его помощи обработка сигнала пользовательской функцией выполняется только один раз, после чего автоматически восстанавливается реакция на сигнал по умолчанию. Для постоянной пользовательской обработки сигнала необходимо каждый раз заново устанавливать реакцию на сигнал прямо внутри функции-обработчика.

В системных вызовах и пользовательских программах могут существовать критические участки, на которых процессу недопустимо отвлекаться на обработку сигналов. Мы можем выставить на этих участках реакцию «игнорировать сигнал» с последующим восстановлением предыдущей реакции, но если сигнал все-таки возникнет на критическом участке, то информация о его возникновении будет безвозвратно потеряна.

Наконец, последний недостаток связан с невозможностью определения количества сигналов одного и того же типа, поступивших процессу, пока он находился в состоянии ***готовность***. Сигналы одного типа в очередь не ставятся! Процесс может узнать о том, что сигнал или сигналы определенного типа были ему переданы, но не может определить их количество. Этот недостаток мы можем проиллюстрировать, слегка изменив программу с асинхронным получением информации о статусе завершившихся процессов, рассмотренную нами ранее в разделе «Изучение

особенностей получения терминальных сигналов текущей и фоновой группой процессов». Пусть в новой программе 12–13-6.c процесс-родитель порождает в цикле пять новых процессов, каждый из которых сразу же завершается со своим собственным кодом, после чего уходит в бесконечный цикл:

```
/* Программа для иллюстрации ненадежности сигналов */
#include <sys/types.h>
#include <unistd.h>
#include <waith>
#include <signal.h>
#include <stdio.h>
/* Функция my_handler - обработчик сигнала SIGCHLD */
void my_handler(int nsig){
  int status;
  pid_t pid;
  /* Опрашиваем статус завершившегося процесса и
  одновременно узнаем его идентификатор */
  if((pid = waitpid(-1, &status, 0)) < 0){
    /* Если возникла ошибка - сообщаем о ней и
    продолжаем работу */
    printf("Some error on waitpid errno = %d\n", errno);
  } else {
    /* Иначе анализируем статус завершившегося процесса */
    if ((status & 0xff) == 0) {
      /* Процесс завершился с явным или неявным
      вызовом функции exit() */
      printf("Process %d was exited with status %d\n",
      pid, status >> 8);
    } else if ((status & 0xff00) == 0){
      /* Процесс был завершен с помощью сигнала */
      printf("Process %d killed by signal %d %s\n",
      pid, status &0x7f,(status & 0x80) ?
      "with core file" : "without core file");
    }
  }
}
int main(void){
  pid_t pid;
  int i;
  /* Устанавливаем обработчик для сигнала SIGCHLD */
  (void) signal(SIGCHLD, my_handler);
```

```
    /* В цикле порождаем пять процессов-детей */
    for (i=0; i
      if((pid = fork()) < 0){
        printf("Can\'t fork child %d\n", i);
        exit(1);
      } else if (pid == 0){
        /* Child i - завершается с кодом 200 + i */
        exit(200 + i);
      }
      /* Продолжение процесса-родителя - уходим на
      новую итерацию */
    }
    /* Продолжение процесса-родителя - уходим в цикл */
    while(1);
    return 0;
}
```

Сколько сообщений о статусе завершившихся детей мы ожидаем получить? Пять! А сколько получим? It depends... Откомпилируйте, прогоните и посчитайте.

Последующие версии System Y и BSD пытались устранить эти недостатки собственными средствами. Единый способ более надежной обработки сигналов появился с введением POSIX-стандарта на системные вызовы UNIX. Набор функций и системных вызовов для работы с сигналами был существенно расширен и построен таким образом, что позволял временно блокировать обработку определенных сигналов, не допуская их потери. Однако проблема, связанная с определением количества пришедших сигналов одного типа, по-прежнему остается актуальной. (Надо отметить, что подобная проблема существует на аппаратном уровне и для внешних прерываний. Процессор зачастую не может определить, какое количество внешних прерываний с одним номером возникло, пока он выполнял очередную команду.)

Рассмотрение POSIX-сигналов выходит за рамки нашего курса. Желающие могут самостоятельно просмотреть описания функций и системных вызовов `sigemptyset()`, `sigfillset()`, `sigaddset()`, `sigdelset()`, `sigismember()`, `sigaction()`, `sigprocmask()`, `sigpending()`, `sigsuspend()` в UNIX Manual.

Задача повышенной сложности: модифицируйте обработку сигнала в программе из этого раздела, не применяя POSIX-сигналы, так, чтобы процесс-родитель все-таки сообщал о статусе всех завершившихся процессов-детей.

Семинары 14–15. Семейство протоколов TCP/IP. Сокеты (sockets) в UNIX и основы работы с ними

Краткая история семейства протоколов TCP/IP. Общие сведения об архитектуре семейства протоколов TCP/IP. Уровень сетевого интерфейса. Уровень Internet. Протоколы IP, ICMP, ARP, RARP. Internet-адреса. Транспортный уровень. Протоколы TCP и UDP, UDP- и TCP-сокеты (sockets). Адресные пространства портов. Понятие encapsulation. Уровень приложений/программ. Использование модели клиент — сервер при изучении сетевого программирования. Организация связи между удаленными процессами с помощью датаграмм. Сетевой порядок байт. Функции `htons()`, `htonl()`, `ntohs()`, `ntohl()`. Функции преобразования IP-адресов `inet_ntoa()`, `inet_aton()`. Функция `bzero()`. Системные вызовы `socket()`, `bind()`, `sendto()`, `recvfrom()`. Организация связи между процессами с помощью установки логического соединения. Системные вызовы `connect()`, `listen()`, `accept()`. Использование интерфейса сокетов для других семейств протоколов. Файлы типа «сокет».

Ключевые слова: семейство протоколов TCP/IP, уровень сетевого интерфейса, уровень Internet, транспортный уровень, уровень приложений/процессов, протоколы сетевого интерфейса, IP, ICMP, ARP, RARP, TCP, UDP, MAC-адрес, IP-адрес, порт, сокет, адрес сокета, encapsulation, датаграмма, виртуальное (логическое соединение), UDP-сокет, TCP-сокет, пассивный (слушающий) TCP-сокет, присоединенный TCP-сокет, трехэтапное рукопожатие, не полностью установленное соединение, полностью установленное соединение, функции `htons()`, `htonl()`, `ntohs()`, `ntohl()`, `inet_ntoa()`, `inet_aton()`, `bzero()`, системные вызовы `socket()`, `bind()`, `sendto()`, `recvfrom()`, `connect()`, `listen()`, `accept()`, UNIX Domain протоколы, файлы типа «сокет».

Краткая история семейства протоколов TCP/IP

Мы приступаем к последней теме наших семинарских и практических занятий – введению в сетевое программирование в операционной системе UNIX.

Все многообразие сетевых приложений и многомиллионная всемирная компьютерная сеть выросли из четырехкомпьютерной сети ARPANET, созданной по заказу Министерства Обороны США и связавшей вычислительные комплексы в Стэндфордском исследовательском институте, Калифорнийском университете в Санта-Барбаре, Калифорний-

ском университете в Лос-Анджелесе и университете Юты. Первая передача информации между двумя компьютерами сети ARPANET состоялась в октябре 1969 года, и эту дату принято считать датой рождения нелокальных компьютерных сетей. (Необходимо отметить, что дата является достаточно условной, так как первая связь двух удаленных компьютеров через коммутируемые телефонные линии была осуществлена еще в 1965 году, а реальные возможности для разработки пользователями ARPANET сетевых приложений появились только в 1972 году.) Эта сеть росла и почковалась, закрывались ее отдельные части, появлялись ее гражданские аналоги, они сливались вместе, и в результате «что выросло — то выросло».

При создании ARPANET был разработан протокол сетевого взаимодействия коммуникационных узлов — Network Control Protocol (NCP), осуществлявший связь посредством передачи датаграмм (см. лекцию 14, раздел «Связь с установлением логического соединения и передача данных с помощью сообщений»). Этот протокол был предназначен для конкретного архитектурного построения сети и базировался на предположении, что сеть является статической и настолько надежной, что компьютерам не требуется умения реагировать на возникающие ошибки. По мере роста ARPANET и необходимости подключения к ней сетей, построенных на других архитектурных принципах (пакетные спутниковые сети, наземные пакетные радиосети), от этого предположения пришлось отказаться и искать другие подходы к построению сетевых систем. Результатом исследований в этих областях стало появление семейства протоколов TCP/IP, на базе которого обеспечивалась надежная доставка информации по неоднородной сети. Это семейство протоколов до сих пор занимает ведущее место в качестве сетевой технологии, используемой в операционной системе UNIX. Именно поэтому мы и выбрали его для практической иллюстрации общих сетевых решений, изложенных в лекции 14.

Общие сведения об архитектуре семейства протоколов TCP/IP

Семейство протоколов TCP/IP построено по «слоеному» принципу, подробно рассмотренному ранее (лекция 14, раздел «Многоуровневая модель построения сетевых вычислительных систем»). Хотя оно и имеет многоуровневую структуру, его строение отличается от строения эталонной модели OSI, предложенной стандартом ISO. Это и неудивительно, так как основные черты семейства TCP/IP были заложены до появления эталонной модели и во многом послужили толчком для ее разработки. В семействе протоколов TCP/IP можно выделить четыре уровня:

1. Уровень сетевого интерфейса.
2. Уровень Internet.

3. Транспортный уровень.
4. Уровень приложений/процессов.

Соотношение уровней модели OSI/ISO и уровней семейства TCP/IP приведено на рисунке 14-15.1.

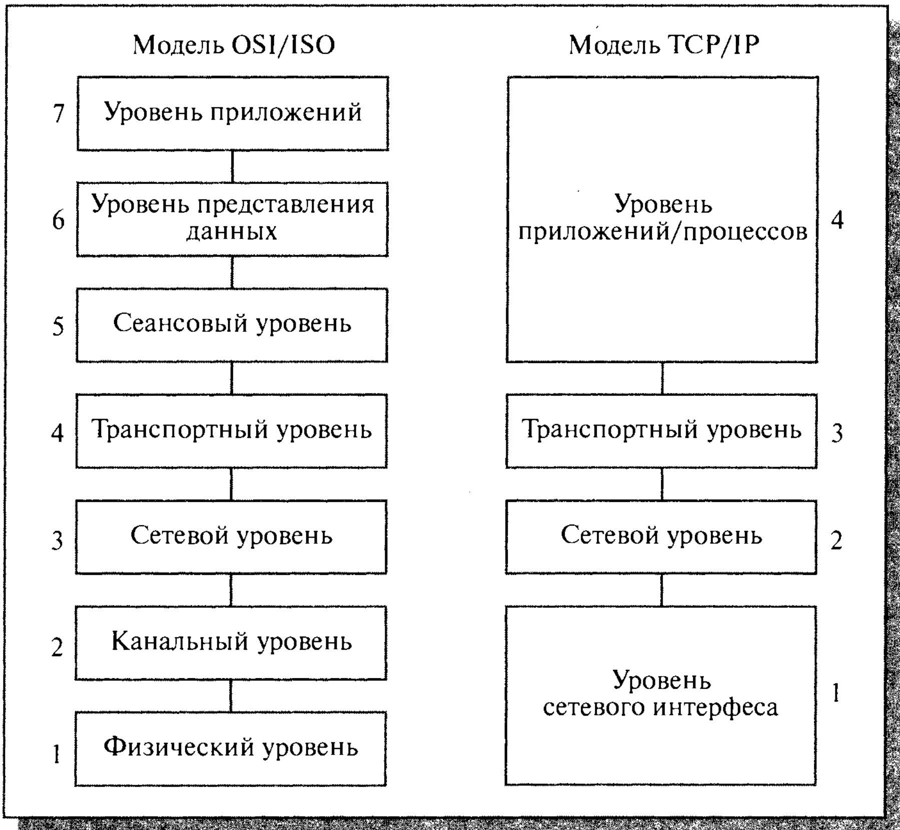


Рис. 14-15.1. Соотношение моделей OSI/ISO и TCP/IP

На каждом уровне семейства TCP/IP присутствует несколько протоколов. Связь между наиболее употребительными протоколами и их принадлежность уровням изображены на рисунке 14-15.2.

Давайте кратко охарактеризуем каждый уровень семейства.

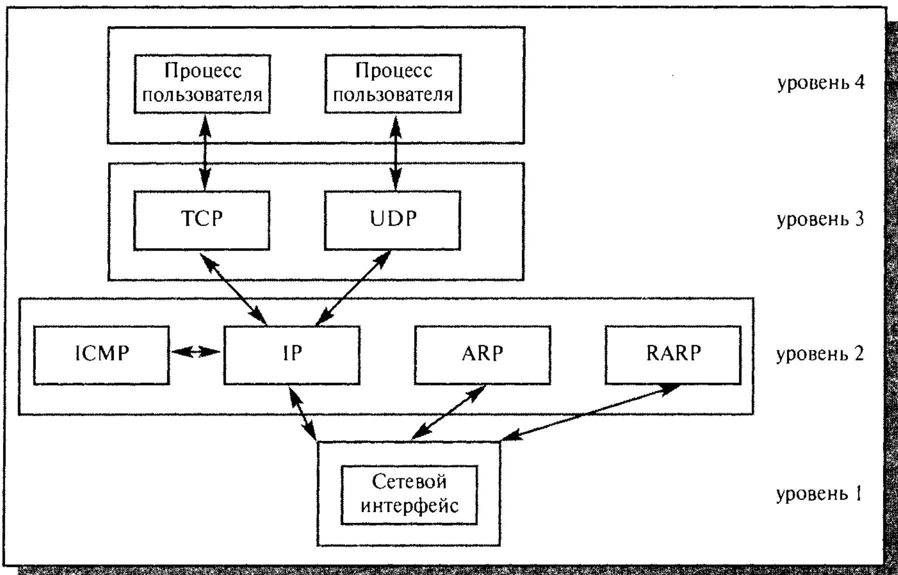


Рис. 14-15.2. Основные протоколы семейства TCP/IP

Уровень сетевого интерфейса

Уровень сетевого интерфейса составляют протоколы, которые обеспечивают передачу данных между узлами связи, физически напрямую соединенными друг с другом, или, иначе говоря, подключенными к одному сегменту сети, и соответствующие физические средства передачи данных. К этому уровню относятся протоколы Ethernet, Token Ring, SLIP, PPP и т. д. и такие физические средства как витая пара, коаксиальный кабель, оптоволоконный кабель и т. д. Формально протоколы уровня сетевого интерфейса не являются частью семейства TCP/IP, но существующие стандарты определяют, каким образом должна осуществляться передача данных семейства TCP/IP с использованием этих протоколов. На уровне сетевого интерфейса в операционной системе UNIX обычно функционируют драйверы различных сетевых плат.

Передача информации на уровне сетевого интерфейса производится на основании физических адресов, соответствующих точкам входа сети в узлы связи (например, физических адресов сетевых карт). Каждая точка входа имеет свой уникальный адрес – MAC-адрес (Media Access Control), физически зашитый в нее на этапе изготовления. Так, например, каждая сетевая плата Ethernet имеет собственный уникальный 48-битовый номер.

Уровень Internet. Протоколы IP, ICMP, ARP, RARP. Internet–адреса

Из многочисленных протоколов уровня Internet мы перечислим только те, которые будут в дальнейшем упоминаться в нашем курсе:

- ICMP – Internet Control Message Protocol. Протокол обработки ошибок и обмена управляющей информацией между узлами сети.
- IP – Internet Protocol. Это протокол, который обеспечивает доставку пакетов информации для протокола ICMP и протоколов транспортного уровня TCP и UDP.
- ARP – Address Resolution Protocol. Это протокол для отображения адресов уровня Internet в адреса уровня сетевого интерфейса.
- RARP – Reverse Address Resolution Protocol. Этот протокол служит для решения обратной задачи: отображения адресов уровня сетевого интерфейса в адреса уровня Internet.

Два последних протокола используются не для всех сетей; только некоторые сети требуют их применения.

Уровень Internet обеспечивает доставку информации от сетевого узла отправителя к сетевому узлу получателя без установления виртуального соединения с помощью датаграмм и не является надежным.

Центральным протоколом уровня является протокол IP. Вся информация, поступающая к нему от других протоколов, оформляется в виде IP-пакетов данных (IP datagrams). Каждый IP-пакет содержит адреса компьютера отправителя и компьютера получателя, поэтому он может передаваться по сети независимо от других пакетов и, возможно, по своему собственному маршруту. Любая ассоциативная связь между пакетами, предполагающая знания об их содержании, должна осуществляться на более высоком уровне семейства протоколов.

IP-уровень семейства TCP/IP не является уровнем, обеспечивающим надежную связь, так как он не гарантирует ни доставку отправленного пакета информации, ни то, что пакет будет доставлен без ошибок. IP вычисляет и проверяет контрольную сумму, которая покрывает только его собственный 20-байтовый заголовок для пакета информации (включающий, например, адреса отправителя и получателя). Если IP-заголовок пакета при передаче оказывается испорченным, то весь пакет просто отбрасывается. Ответственность за повторную передачу пакета тем самым возлагается на вышестоящие уровни.

IP-протокол, при необходимости, осуществляет фрагментацию и дефрагментацию данных, передаваемых по сети. Если размер IP-пакета слишком велик для дальнейшей передачи по сети, то полученный пакет разбивается на несколько фрагментов, и каждый фрагмент оформляется в виде нового IP-пакета с теми же адресами отправителя и получателя.

Фрагменты собираются в единое целое только в конечной точке своего путешествия. Если при дефрагментации пакета обнаруживается, что хотя бы один из фрагментов был потерян или отброшен, то отбрасывается и весь пакет целиком.

Уровень Internet отвечает за маршрутизацию пакетов. Для обмена информацией между узлами сети в случае возникновения проблем с маршрутизацией пакетов используется протокол ICMP. С помощью сообщений этого же протокола уровень Internet умеет частично управлять скоростью передачи данных – он может попросить отправителя уменьшить скорость передачи.

Поскольку на уровне Internet информация передается от компьютера-отправителя к компьютеру-получателю, ему требуются специальные IP-адреса компьютеров (а точнее, их точек подсоединения к сети – сетевых интерфейсов) – удаленные части полных адресов процессов (см. лекцию 14, раздел «Удаленная адресация и разрешение адресов»). Мы будем далее работать с IP версии 4 (IPv4), которая предполагает наличие у каждого сетевого интерфейса уникального 32-битового адреса. Когда разрабатывалось семейство протоколов TCP/IP, казалось, что 32 битов адреса будет достаточно для всех нужд сети, однако не прошло и 30 лет, как выяснилось, что этого мало. Поэтому была разработана версия 6 для IP (IPv6), предполагающая наличие 128-битовых адресов. С точки зрения сетевого программиста IPv6 мало отличается от IPv4, но имеет более сложный интерфейс передачи параметров, поэтому для практических занятий был выбран IPv4.

Все IP-адреса версии 4 принято делить на 5 классов. Принадлежность адреса к некоторому классу определяют по количеству последовательных едениц в старших битах адреса (см. рис. 14-15.3). Адреса классов A, B и C используют собственно для адресации сетевых интерфейсов. Адреса класса D применяются для групповой рассылки информации (multicast addresses) и далее нас интересовать не будут. Класс E (про который во многих книгах по сетям забывают) был зарезервирован для будущих расширений.

Каждый из IP-адресов классов A–C логически делится на две части: идентификатор или номер сети и идентификатор или номер узла в этой сети. Идентификаторы сетей в настоящее время присваиваются локальным сетям специальной международной организацией – корпорацией Internet по присвоению имен и номеров (ICANN). Присвоение адреса конкретному узлу сети, получившей идентификатор, является заботой ее администратора. Класс A предназначен для небольшого количества сетей, содержащих очень много компьютеров, класс C – напротив, для большого количества сетей с малым числом компьютеров. Класс B занимает среднее положение. Надо отметить, что все идентификаторы сетей классов A и B к настоящему моменту уже задействованы.

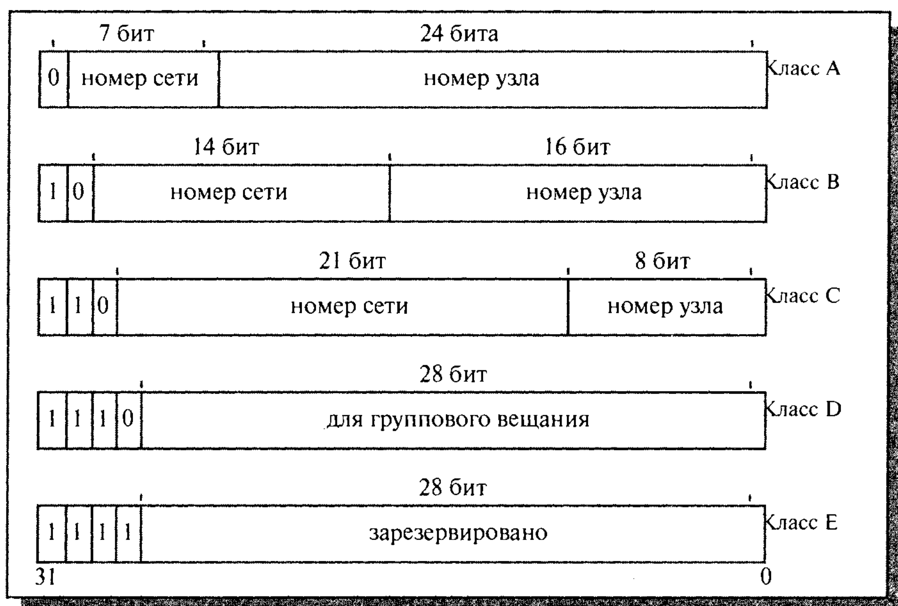


Рис. 14-15.3. Классы IP-адресов

Любая организация, которой был выделен идентификатор сети из любого класса, может произвольным образом разделить имеющееся у нее адресное пространство идентификаторов узлов для создания подсетей.

Допустим, что вам выделен адрес сети класса С, в котором под номер узла сети отведено 8 бит. Если нужно присвоить IP-адреса 100 компьютерам, которые организованы в 10 Ethernet-сегментов по 10 компьютеров в каждом, можно поступить по-разному. Можно присвоить компьютерам номера от 1 до 100, игнорируя их принадлежность к конкретному сегменту – воспользовавшись стандартной формой IP-адреса. Или же можно выделить несколько младших бит из адресного пространства идентификаторов узлов для идентификации сегмента сети, например 4 бита, а для адресации узлов внутри сегмента использовать оставшиеся 4 бита. Последний способ получил название адресации с использованием подсетей (см. рис. 14-15.4).

Запоминать четырехбайтовые числа для человека достаточно сложно, поэтому принято записывать IP-адреса в символической форме, переводя значение каждого байта в десятичный вид по отдельности и разделяя полученные десятичные числа в записи точками, начиная со старшего байта: **192.168.253.10**.

Допустим, что мы имеем дело с сегментом сети, использующим Ethernet на уровне сетевого интерфейса и состоящим из компьютеров, где

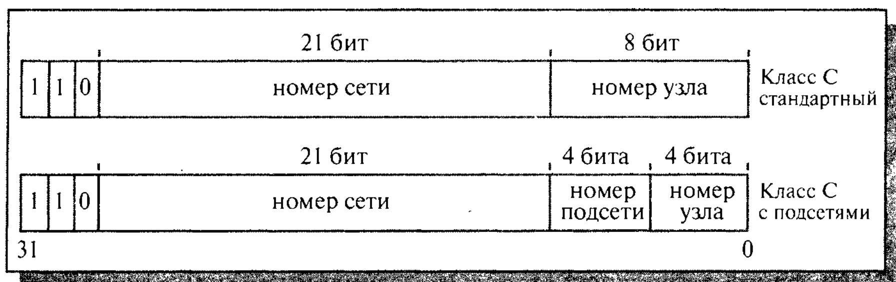


Рис. 14-15.4. Адресация с подсетями

применяются протоколы TCP/IP на более высоких уровнях. Тогда у нас в сети есть два вида адресов: 48-битовые физические адреса Ethernet (MAC-адреса) и 32-битовые IP-адреса. Для нормальной передачи информации необходимо, чтобы Internet-уровень семейства протоколов, обращаясь к уровню сетевого интерфейса, знал, какой физический адрес соответствует данному IP-адресу и наоборот, т. е. умел «разрешать адреса». В очередной раз мы сталкиваемся с проблемой разрешения адресов, которая в различных постановках разбиралась в материалах лекций. При разрешении адресов могут возникнуть две сложности:

- Если мы знаем IP-адреса компьютеров, которым или через которые мы хотим передать данные, то каким образом Internet уровень семейства протоколов TCP/IP сможет определить соответствующие им MAC-адреса? Эта проблема получила название address resolution problem (проблема разрешения адресов).
- Пусть у нас есть бездисковые рабочие станции или рабочие станции, на которых операционные системы сгенерированы без назначения IP-адресов (это часто делается, когда один и тот же образ операционной системы загружается на ряд компьютеров, например, в учебных классах). Тогда при старте операционной системы на каждом таком компьютере операционная система знает только MAC-адреса, соответствующие данному компьютеру. Как можно определить, какой Internet-адрес был выделен данной рабочей станции? Эта проблема называется reverse address resolution problem (обратная проблема разрешения адресов).

Первая задача решается с использованием протокола ARP, вторая – с помощью протокола RARP.

Протокол ARP позволяет компьютеру разослать специальное сообщение по всему сегменту сети, которое требует от компьютера, имеющего содержащийся в сообщении IP-адрес, откликнуться и указать свой физический адрес. Это сообщение поступает всем компьютерам в сегменте сети, но откликается на него только тот, кого спрашивали. После получения

ответа запрашивавший компьютер может установить необходимое соответствие между IP-адресом и MAC-адресом.

Для решения второй проблемы один или несколько компьютеров в сегменте сети должны выполнять функции RARP-сервера и содержать набор физических адресов для рабочих станций и соответствующих им IP-адресов. Когда рабочая станция с операционной системой, сгенерированной без назначения IP-адреса, начинает свою работу, она получает MAC-адрес от сетевого оборудования и рассылает соответствующий RARP-запрос, содержащий этот адрес, всем компьютерам сегмента сети. Только RARP-сервер, содержащий информацию о соответствии указанного физического адреса и выделенного IP-адреса, откликается на данный запрос и отправляет ответ, содержащий IP-адрес.

Транспортный уровень. Протоколы TCP и UDP. TCP- и UDP-сокеты. Адресные пространства портов. Понятие encapsulation

Мы не будем вдаваться в детали реализации протоколов транспортного уровня, а лишь кратко рассмотрим их основные характеристики. К протоколам транспортного уровня относятся протоколы TCP и UDP.

Протокол TCP реализует потоковую модель передачи информации, хотя в его основе, как и в основе протокола UDP, лежит обмен информацией через пакеты данных. Он представляет собой ориентированный на установление логической связи (connection-oriented), надежный (обеспечивающий проверку контрольных сумм, передачу подтверждения в случае правильного приема сообщения, повторную передачу пакета данных в случае неполучения подтверждения в течение определенного промежутка времени, правильную последовательность получения информации, полный контроль скорости передачи данных) дуплексный способ связи между процессами в сети. Протокол UDP, наоборот, является ненадежным способом связи, ориентированным на передачу сообщений (датаграмм). От протокола IP он отличается двумя основными чертами: использованием для проверки правильности принятого сообщения контрольной суммы, насчитанной по всему сообщению, и передачей информации не от узла сети к другому узлу, а от отправителя к получателю.

На лекции 14 (раздел «Полные адреса. Понятие сокета (socket)») мы говорили, что полный адрес удаленного процесса или промежуточного объекта для конкретного способа связи с точки зрения операционных систем определяется парой адресов: <числовой адрес компьютера в сети, локальный адрес>.

Такая пара получила название socket (гнездо, панель), так как по сути дела является виртуальным коммуникационным узлом (можно представить себе виртуальный разъем или ящик для приема/отправки писем), ведущим от объекта во внешний мир и наоборот. При непрямой адресации сами промежуточные объекты для организации взаимодействия процессов также именуются сокетами.

Поскольку уровень Internet семейства протоколов TCP/IP умеет доставлять информацию только от компьютера к компьютеру, данные, полученные с его помощью, должны содержать тип использованного протокола транспортного уровня и локальные адреса отправителя и получателя. И протокол TCP, и протокол UDP используют непрямую адресацию.

Для того чтобы избежать путаницы, мы в дальнейшем термин «сокет» будем употреблять только для обозначения самих промежуточных объектов, а полные адреса таких объектов будем называть адресами сокетов.

Для каждого транспортного протокола в стеке TCP/IP существуют собственные сокеты: UDP- сокеты и TCP-сокеты, имеющие различные адресные пространства своих локальных адресов – портов. В семействе протоколов TCP/IP адресные пространства портов представляют собой положительные значения целого 16-битового числа. Поэтому, говоря о локальном адресе сокета, мы часто будем использовать термин «*номер порта*». Из различия адресных пространств портов следует, что порт 1111 TCP – это совсем не тот же самый локальный адрес, что и порт 1111 UDP. О том, как назначаются номера портов различным сокетам, мы поговорим позже.

Итак, мы описали иерархическую систему адресации, используемую в семействе протоколов TCP/IP, которая включает в себя несколько уровней:

- Физический пакет данных, передаваемый по сети, содержит физические адреса узлов сети (MAC-адреса) с указанием на то, какой протокол уровня Internet должен использоваться для обработки передаваемых данных (поскольку пользователя интересуют только данные, доставляемые затем на уровень приложений/процессов, то для него это всегда IP).
- IP-пакет данных содержит 32-битовые IP-адреса компьютера-отправителя и компьютера-получателя, и указание на то, какой вышележащий протокол (TCP, UDP или еще что-нибудь) должен использоваться для их дальнейшей обработки.
- Служебная информация транспортных протоколов (UDP-заголовок к данным и TCP-заголовок к данным) должна содержать 16-битовые номера портов для сокета отправителя и сокета получателя.

Добавление необходимой информации к данным при переходе от верхних уровней семейства протоколов к нижним принято называть английским словом encapsulation (дословно: герметизация). На рисунке 14-15.5 приведена схема encapsulation при использовании протокола UDP на сети Ethernet.

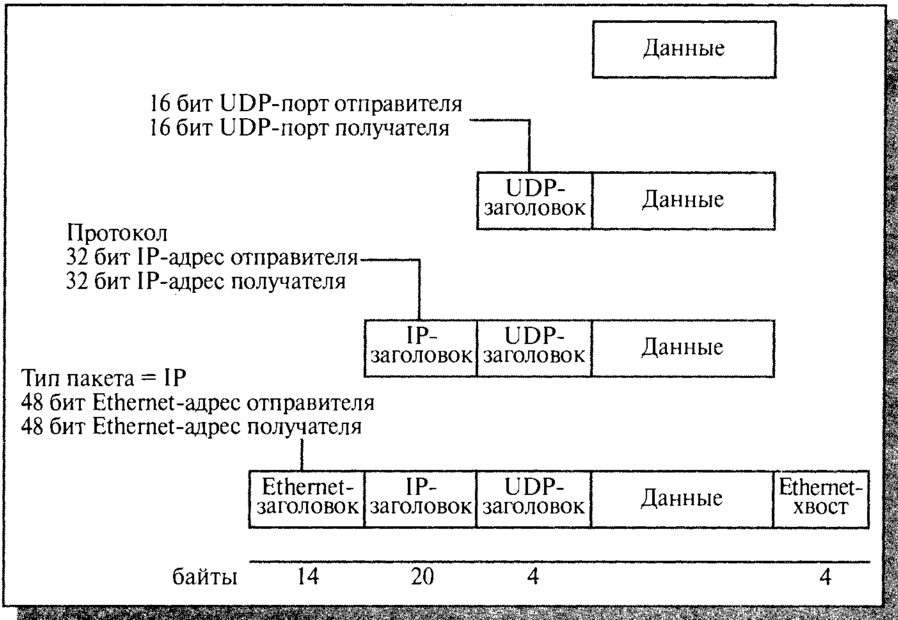


Рис. 14-15.5. Encapsulation для UDP-протокола на сети Ethernet

Поскольку между MAC-адресами и IP-адресами существует взаимно однозначное соответствие, известное семейству протоколов TCP/IP, то фактически для полного задания адреса доставки и адреса отправления, необходимых для установления двусторонней связи, нужно указать пять параметров:

<транспортный протокол, IP-адрес отправителя, порт отправителя, IP-адрес получателя, порт получателя>.

Уровень приложений/процессов

К этому уровню можно отнести протоколы TFTP (**T**rivial **F**ile **T**ransfer **P**rotocol), FTP (**F**ile **T**ransfer **P**rotocol), telnet, SMTP (**S**imple **M**ail **T**ransfer **P**rotocol) и другие, которые поддерживаются соответствующими системными утилитами. Об их использовании подробно рассказано в UNIX Manual, и останавливаться на них мы не будем.

Нас будет интересовать в дальнейшем программный интерфейс между уровнем приложений/процессов и транспортным уровнем для того, чтобы мы могли создавать собственные процессы, общающиеся через сеть. Но прежде чем заняться программным интерфейсом, нам необходимо вспомнить особенности взаимодействия процессов в модели клиент–сервер.

Использование модели клиент–сервер для взаимодействия удаленных процессов

В материалах семинара 9 при обсуждении мультиплексирования сообщений (раздел «Понятие мультиплексирования. Мультиплексирование сообщений. Модель взаимодействия процессов клиент–сервер. Неравноправность клиента и сервера») говорилось об использовании модели клиент — сервер для организации взаимодействия локальных процессов. Эта же модель, изначально предполагающая неравноправность взаимодействующих процессов, наиболее часто используется для организации сетевых приложений. Напомним основные отличия процессов клиента и сервера применительно к удаленному взаимодействию:

- Сервер, как правило, работает постоянно, на всем протяжении жизни приложения, а клиенты могут работать эпизодически.
- Сервер ждет запроса от клиентов, инициатором же взаимодействия выступает клиент.
- Как правило, клиент обращается к одному серверу за раз, в то время как к серверу могут одновременно поступить запросы от нескольких клиентов.
- Клиент должен знать полный адрес сервера (его локальную и удаленную части) перед началом организации запроса (до начала общения), в то время как сервер может получить информацию о полном адресе клиента из пришедшего запроса (после начала общения).
- И клиент, и сервер должны использовать один и тот же протокол транспортного уровня.

Неравноправность процессов в модели клиент–сервер, как мы увидим далее, накладывает свой отпечаток на программный интерфейс, используемый между уровнем приложений/процессов и транспортным уровнем.

Поступающие запросы сервер может обрабатывать последовательно – запрос за запросом – или параллельно, запуская для обработки каждого из них свой процесс или thread. Как правило, серверы, ориентированные на связь клиент–сервер с помощью установки логического соединения (TCP-протокол), ведут обработку запросов параллельно, а серверы, ориентированные на связь клиент–сервер без установления соединения (UDP-протокол), обрабатывают запросы последовательно.

Рассмотрим основные действия, которые нам необходимы в терминах абстракции socket для того, чтобы организовать взаимодействие между клиентом и сервером, используя транспортные протоколы стека TCP/IP.

Организация связи между удаленными процессами с помощью датаграмм

Как уже упоминалось в лекциях, более простой для взаимодействия удаленных процессов является схема организации общения клиента и сервера с помощью датаграмм, т. е. использование протокола UDP.

Рассмотрение этой схемы мы начнем с некоторой житейской аналогии, а затем убедимся, что каждому житейски обоснованному действию в операционной системе UNIX соответствует определенный системный вызов.

С точки зрения обычного человека общение процессов посредством датаграмм напоминает общение людей в письмах. Каждое письмо представляет собой законченное сообщение, содержащее адрес получателя, адрес отправителя и указания, кто написал письмо и кто должен его получить. Письма могут теряться, доставляться в неправильном порядке, быть поврежденными в дороге и т. д.

Что в первую очередь должен сделать человек, проживающий в отдаленной местности, для того чтобы принимать и отправлять письма? Он должен изготовить почтовый ящик, который одновременно будет служить и для приема корреспонденции, и для ее отправки. Пришедшие письма почтальон будет помещать в этот ящик и забирать из него письма, подготовленные к отправке.

Изготовленный почтовый ящик нужно где-то прикрепить. Это может быть парадная дверь дома или вход со двора, изгородь, столб, дерево и т. п. Потенциально может быть изготовлено несколько почтовых ящиков и размещено в разных местах с тем, чтобы письма от различных адресатов прибывали в различные ящики. Этим ящикам будут соответствовать разные адреса: «г. Иванову, почтовый ящик на конюшне», «г. Иванову, почтовый ящик, что на дубе».

После закрепления ящика мы готовы к обмену корреспонденцией. Человек-клиент пишет письмо с запросом по заранее известному ему адресу человека-сервера и ждет получения ответного письма. После получения ответа он читает его и перерабатывает полученную информацию.

Человек-сервер изначально находится в состоянии ожидания запроса. Получив письмо, он читает текст запроса и определяет адрес отправителя. После обработки запроса он пишет ответ и отправляет его по обратному адресу, после чего начинает ждать следующего запроса.

Все эти модельные действия имеют аналоги при общении удаленных процессов по протоколу UDP.

Процесс-сервер должен сначала совершить подготовительные действия: создать UDP-сокет (изготовить почтовый ящик) и связать его с определенным номером порта и IP-адресом сетевого интерфейса (прикрепить почтовый ящик в определенном месте) – настроить адрес сокета. При

этом сокет может быть привязан к конкретному сетевому интерфейсу (к конюшне, к дубу) или к компьютеру в целом, то есть в полном адресе сокета может быть либо указан IP-адрес конкретного сетевого интерфейса, либо дано указание операционной системе, что информация может поступить через любой сетевой интерфейс, имеющийся в наличии. После настройки адреса сокета операционная система начинает принимать сообщения, пришедшие на этот адрес, и складывать их в сокет. Сервер дожидается поступления сообщения, читает его, определяет, от кого оно поступило и через какой сетевой интерфейс, обрабатывает полученную информацию и отправляет результат по обратному адресу. После чего процесс готов к приему новой информации от того же или другого клиента.

Процесс-клиент должен сначала совершить те же самые подготовительные действия: создать сокет и настроить его адрес. Затем он передает сообщение, указав, кому оно предназначено (IP-адрес сетевого интерфейса и номер порта сервера), ожидает от него ответа и продолжает свою деятельность.

Схематично эти действия выглядят так, как показано на рисунке 15-16.6. Каждому из них соответствует определенный системный вызов. Названия вызовов написаны справа от блоков соответствующих действий.

Создание сокета производится с помощью системного вызова `socket()`. Для привязки созданного сокета к IP-адресу и номеру порта (настройка адреса) служит системный вызов `bind()`. Ожиданию получения информации, ее чтению и, при необходимости, определению адреса отправителя соответствует системный вызов `recvfrom()`. За отправку датаграммы отвечает системный вызов `sendto()`.

Прежде чем приступить к подробному рассмотрению этих системных вызовов и примеров программ, нам придется остановиться на нескольких вспомогательных функциях, которые мы должны будем использовать при программировании.

Сетевой порядок байт. Функции htons(), htonl(), ntohs(), ntohl()

Передача от одного вычислительного комплекса к другому символьной информации, как правило (когда один символ занимает один байт), не вызывает проблем. Однако для числовой информации ситуация усложняется.

Как известно, порядок байт в целых числах, представление которых занимает более одного байта, может быть для различных компьютеров неодинаковым. Есть вычислительные системы, в которых старший байт числа имеет меньший адрес, чем младший байт (big-endian

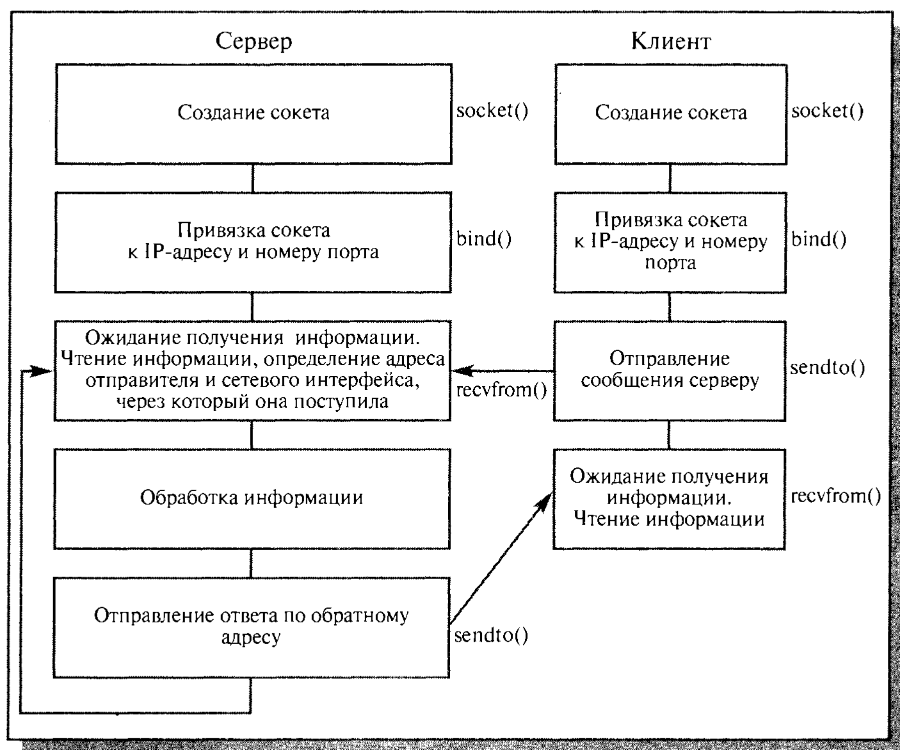


Рис. 14-15.6. Схема взаимодействия клиента и сервера для протокола UDP

byte order), а есть вычислительные системы, в которых старший байт числа имеет больший адрес, чем младший байт (little-endian byte order). При передаче целой числовой информации от машины, имеющей один порядок байт, к машине с другим порядком байт мы можем неправильно истолковать принятую информацию. Для того чтобы этого не произошло, было введено понятие сетевого порядка байт, т. е. порядка байт, в котором должна представляться целая числовая информация в процессе передачи ее по сети (на самом деле – это big-endian byte order). Целые числовые данные из представления, принятого на компьютере-отправителе, переводятся пользовательским процессом в сетевой порядок байт, путешествуют в таком виде по сети и переводятся в нужный порядок байт на машине-получателе процессом, которому они предназначены. Для перевода целых чисел из машинного представления в сетевое и обратно используется четыре функции: `htons()`, `htonl()`, `ntohs()`, `ntohl()`.

Функции преобразования порядка байт

Прототипы функций

```
#include <netinet/in.h>
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

Описание функций

Функция `htonl` осуществляет перевод целого длинного числа из порядка байт, принятого на компьютере, в сетевой порядок байт.

Функция `htons` осуществляет перевод целого короткого числа из порядка байт, принятого на компьютере, в сетевой порядок байт.

Функция `ntohl` осуществляет перевод целого длинного числа из сетевого порядка байт в порядок байт, принятый на компьютере.

Функция `ntohs` осуществляет перевод целого короткого числа из сетевого порядка байт в порядок байт, принятый на компьютере.

В архитектуре компьютеров i80x86 принят порядок байт, при котором младшие байты целого числа имеют младшие адреса. При сетевом порядке байт, принятом в Internet, младшие адреса имеют старшие байты числа.

Параметр у них – значение, которое мы собираемся конвертировать. Возвращаемое значение – то, что получается в результате конвертации. Направление конвертации определяется порядком букв h (host) и n (network) в названии функции, размер числа – последней буквой названия, то есть `htons` – это **h**ost **to** **n**etwork **s**hort, `ntohl` – **n**etwork **to** **h**ost **l**ong.

Для чисел с плавающей точкой все обстоит гораздо хуже. На разных машинах могут различаться не только порядок байт, но и форма представления такого числа. Простых функций для их корректной передачи по сети не существует. Если требуется обмениваться действительными данными, то это нужно делать либо на гомогенной сети, состоящей из одинаковых компьютеров, либо использовать символьные и целые данные для передачи действительных значений.

Функции преобразования IP-адресов inet_ntoa(), inet_aton()

Нам также понадобятся функции, осуществляющие перевод IP-адресов из символьного представления (в виде четверки чисел, разделенных точками) в числовое представление и обратно. Функция `inet_aton()`

переводит символьный IP-адрес в числовое представление в сетевом порядке байт.

Функция возвращает 1, если в символьном виде записан правильный IP-адрес, и 0 в противном случае – **для большинства системных вызовов и функций это нетипичная ситуация. Обратите внимание на использование указателя на структуру `struct in_addr` в качестве одного из параметров данной функции. Эта структура используется для хранения IP-адресов в сетевом порядке байт. То, что используется структура, состоящая из одной переменной, а не сама 32-битовая переменная, сложилось исторически, и авторы в этом не виноваты.**

Для обратного преобразования применяется функция `inet_ntoa()`.

Функции преобразования IP-адресов

Прототипы функций

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
int inet_aton(const char *strptr,
  struct in_addr *addrptr);
char *inet_ntoa(struct in_addr *addrptr);
```

Описание функций

Функция `inet_aton` переводит символьный IP-адрес, расположенный по указателю `strptr`, в числовое представление в сетевом порядке байт и заносит его в структуру, расположенную по адресу `addrptr`. Функция возвращает значение 1, если в строке записан правильный IP-адрес, и значение 0 в противном случае. Структура типа `struct in_addr` используется для хранения IP-адресов в сетевом порядке байт и выглядит так:

```
struct in_addr {
  in_addr_t s_addr;
};
```

То, что используется адрес такой структуры, а не просто адрес переменной типа `in_addr_t`, сложилось исторически.

Функция `inet_ntoa` применяется для обратного преобразования. Числовое представление адреса в сетевом порядке байт должно быть занесено в структуру типа struct `in_addr`, адрес которой `addrptr` передается функции как аргумент. Функция возвращает указатель на строку, содержащую символьное представление адреса. Эта строка располагается в статическом буфере, при последующих вызовах ее новое содержимое заменяет старое содержимое.

Функция bzero()

Функция bzero настолько проста, что про нее нечего рассказывать. Все видно из описания.

Функция bzero

Прототип функции

```
#include <string.h>
void bzero(void *addr, int n);
```

Описание функции

Функция bzero заполняет первые n байт, начиная с адреса addr, нулевыми значениями. Функция ничего не возвращает.

Теперь мы можем перейти к системным вызовам, образующим интерфейс между пользовательским уровнем стека протоколов TCP/IP и транспортным протоколом UDP.

Создание сокета. Системный вызов socket()

Для создания сокета в операционной системе служит системный вызов `socket()`. Для транспортных протоколов семейства TCP/IP существует два вида сокетов: UDP-сокет – сокет для работы с датаграммами, и TCP-сокет – потоковый сокет. Однако понятие сокета (см. лекцию 14, раздел «Полные адреса. Понятие сокета (socket)») не ограничивается рамками только этого семейства протоколов. Рассматриваемый интерфейс сетевых системных вызовов (`socket()`, `bind()`, `recvfrom()`, `sendto()` и т. д.) в операционной системе UNIX может применяться и для других стеков протоколов (и для протоколов, лежащих ниже транспортного уровня).

При создании сокета необходимо точно специфицировать его тип. Эта спецификация производится с помощью трех параметров вызова `socket()`. Первый параметр указывает, к какому семейству протоколов относится создаваемый сокет, а второй и третий параметры определяют конкретный протокол внутри данного семейства.

Второй параметр служит для задания вида интерфейса работы с сокетом – будь это потоковый сокет, сокет для работы с датаграммами или какой-либо иной. Третий параметр указывает протокол для заданного типа интерфейса. В стеке протоколов TCP/IP существует только один протокол для потоковых сокетов – TCP и только один протокол для да-

таграммных сокетов – UDP, поэтому для транспортных протоколов TCP/IP третий параметр игнорируется.

В других стеках протоколов может быть несколько протоколов с одинаковым видом интерфейса, например, датаграммных, различающихся по степени надежности.

Для транспортных протоколов TCP/IP мы всегда в качестве первого параметра будем указывать предопределенную константу AF_INET (Address family – Internet) или ее синоним PF_INET (Protokol family – Internet).

Второй параметр будет принимать предопределенные значения SOCK_STREAM для потоковых сокетов и SOCK_DGRAM – для датаграммных.

Поскольку третий параметр в нашем случае не учитывается, в него мы будем подставлять значение 0.

Ссылка на информацию о созданном сокете помещается в таблицу открытых файлов процесса подобно тому, как это делалось для pip'ов и FIFO (см. семинар 5). Системный вызов возвращает пользователю файловый дескриптор, соответствующий заполненному элементу таблицы, который далее мы будем называть дескриптором сокета. Такой способ хранения информации о сокете позволяет, во-первых, процессам-детям наследовать ее от процессов-родителей, а во-вторых — использовать для сокетов часть системных вызовов, которые уже знакомы нам по работе с pip'ами и FIFO: close(), read(), write().

Системный вызов для создания сокета

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Описание системного вызова

Системный вызов socket служит для создания виртуального коммуникационного узла в операционной системе. Данное описание не является полным описанием системного вызова, а предназначено только для использования в нашем курсе. За полной информацией обращайтесь к UNIX Manual.

Параметр domain определяет семейство протоколов, в рамках которого будет осуществляться передача информации. Мы рассмотрим только два таких семейства из нескольких существующих. Для них имеются предопределенные значения параметра:

PF_INET – для семейства протоколов TCP/IP;

PF_UNIX – для семейства внутренних протоколов UNIX, иначе называемого еще UNIX domain.

Параметр type определяет семантику обмена информацией: будет ли осуществляться связь через сообщения (datagrams), с помощью установления виртуального соединения или еще каким-либо способом. Мы будем пользоваться только двумя способами обмена информацией с предопределенными значениями для параметра type:

SOCK_STREAM – для связи с помощью установления виртуального соединения;

SOCK_DGRAM – для обмена информацией через сообщения.

Параметр protocol специфицирует конкретный протокол для выбранного семейства протоколов и способа обмена информацией. Он имеет значение только в том случае, когда таких протоколов существует несколько. В нашем случае семейство протоколов и тип обмена информацией определяют протокол однозначно. Поэтому данный параметр мы будем полагать равным 0.

Возвращаемое значение

В случае успешного завершения системный вызов возвращает файловый дескриптор (значение большее или равное 0), который будет использоваться как ссылка на созданный коммуникационный узел при всех дальнейших сетевых вызовах. При возникновении какой-либо ошибки возвращается отрицательное значение.

Адреса сокетов. Настройка адреса сокета. Системный вызов bind()

Когда сокет создан, необходимо настроить его адрес. Для этого используется системный вызов bind(). Первый параметр вызова должен содержать дескриптор сокета, для которого производится настройка адреса. Второй и третий параметры задают этот адрес.

Во втором параметре должен быть указатель на структуру struct sockaddr, содержащую удаленную и локальные части полного адреса.

Указатели типа struct sockaddr* встречаются во многих сетевых системных вызовах; они используются для передачи информации о том, к какому адресу привязан или должен быть привязан сокет. Рассмотрим этот тип данных подробнее. Структура struct sockaddr описана в файле <sys/socket.h> следующим образом:

```
struct sockaddr {
  short sa_family;
  char sa_data[14];
};
```

Такой состав структуры обусловлен тем, что сетевые системные вызовы могут применяться для различных семейств протоколов, которые по-разному определяют адресные пространства для удаленных и локальных адресов сокета. По сути дела, этот тип данных представляет собой лишь общий шаблон для передачи системным вызовам структур данных,

специфических для каждого семейства протоколов. Общим элементом этих структур остается только поле short sa_family (которое в разных структурах, естественно, может иметь разные имена, важно лишь, чтобы все они были одного типа и были первыми элементами своих структур) для описания семейства протоколов. Системный вызов анализирует содержимое этого поля для точного определения состава поступившей информации.

Для работы с семейством протоколов TCP/IP мы будем использовать адрес сокета следующего вида, описанного в файле <netinet/in.h>:

```
struct sockaddr _in{
   short sin_family; /* Избранное семейство протоколов
      - всегда AF_INET */
   unsigned short sin_port; /* 16-битовый номер порта
      в сетевом порядке байт */
   struct in_addr sin_addr; /* Адрес сетевого
      интерфейса */
   char sin_zero[8]; /* Это поле не используется,
      но должно всегда быть заполнено нулями */
};
```

Первый элемент структуры – sin_family – задает семейство протоколов. В него мы будем заносить уже известную нам предопределенную константу AF_INET (см. предыдущий раздел).

Удаленная часть полного адреса – IP-адрес – содержится в структуре типа struct in_addr, с которой мы встречались в разделе «Функции преобразования IP-адресов inet_ntoa(), inet_aton()» .

Для указания номера порта предназначен элемент структуры sin_port, в котором номер порта должен храниться **в сетевом порядке байт**. Существует два варианта задания номера порта: фиксированный порт по желанию пользователя и порт, который произвольно назначает операционная система. Первый вариант требует указания в качестве номера порта положительного, заранее известного числа, и для протокола UDP обычно используется при настройке адресов сокетов и при передаче информации с помощью системного вызова sendto() (см. следующий раздел). Второй вариант требует указания в качестве номера порта значения 0. В этом случае операционная система сама привязывает сокет к свободному номеру порта. Этот способ обычно используется при настройке сокетов программ клиентов, когда заранее точно знать номер порта для программиста необязательно.

Какой номер порта может задействовать пользователь при фиксированной настройке? Номера портов с 1 по 1023 могут назначать сокетам

только процессы, работающие с привилегиями системного администратора. Как правило, эти номера закреплены за системными сетевыми службами независимо от вида используемой операционной системы, для того чтобы пользовательские клиентские программы могли запрашивать обслуживание всегда по одним и тем же локальным адресам. Существует также ряд широко применяемых сетевых программ, которые запускают процессы с полномочиями обычных пользователей (например, X-Windows). Для таких программ корпорацией Internet по присвоению имен и номеров (ICANN) выделяется диапазон адресов с 1024 по 49151, который нежелательно использовать во избежание возможных конфликтов. Номера портов с 49152 по 65535 предназначены для процессов обычных пользователей. Во всех наших примерах при фиксированном задании номера порта у сервера мы будем использовать номер 51000.

IP-адрес при настройке также может быть определен двумя способами. Он может быть привязан к конкретному сетевому интерфейсу (т. е. сетевой плате), заставляя операционную систему принимать/передавать информацию только через этот сетевой интерфейс, а может быть привязан и ко всей вычислительной системе в целом (информация может быть получена/отослана через любой сетевой интерфейс). В первом случае в качестве значения поля структуры `sin_addr.s_addr` используется числовое значение IP-адреса конкретного сетевого интерфейса в сетевом порядке байт. Во втором случае это значение должно быть равно значению предопределенной константы `INADDR_ANY`, приведенному к сетевому порядку байт.

Третий параметр системного вызова `bind()` должен содержать фактическую длину структуры, адрес которой передается в качестве второго параметра. Эта длина меняется в зависимости от семейства протоколов и даже различается в пределах одного семейства протоколов. Размер структуры, содержащей адрес сокета, для семейства протоколов TCP/IP может быть определен как `sizeof(struct sockaddr_in)`.

Системный вызов для привязки сокета к конкретному адресу

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockd, struct sockaddr *my_addr,
   int addrlen);
```

Описание системного вызова

Системный вызов `bind` служит для привязки созданного сокета к определенному полному адресу вычислительной сети.

Параметр `sockd` является дескриптором созданного ранее коммуникационного узла, т. е. значением, которое вернул системный вызов `socket()`.

Параметр `my_addr` представляет собой адрес структуры, содержащей информацию о том, куда именно мы хотим привязать наш сокет – то, что принято называть адресом сокета. Он имеет тип указателя на структуру-шаблон `struct sockaddr`, которая должна быть конкретизирована в зависимости от используемого семейства протоколов и заполнена перед вызовом.

Параметр `addrlen` должен содержать фактическую длину структуры, адрес которой передается в качестве второго параметра. Эта длина в разных семействах протоколов и даже в пределах одного семейства протоколов может быть различной (например, для UNIX Domain).

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и отрицательное значение – в случае ошибки.

Системные вызовы sendto() и recvfrom()

Для отправки датаграмм применяется системный вызов `sendto()`. В число параметров этого вызова входят:

- дескриптор сокета, через который отсылается датаграмма;
- адрес области памяти, где лежат данные, которые должны составить содержательную часть датаграммы, и их длина;
- флаги, определяющие поведение системного вызова (в нашем случае они всегда будут иметь значение 0);
- указатель на структуру, содержащую адрес сокета получателя, и ее фактическая длина.

Системный вызов возвращает отрицательное значение при возникновении ошибки и количество реально отосланных байт при нормальной работе. **Нормальное завершение системного вызова не означает, что датаграмма уже покинула ваш компьютер!** Датаграмма сначала помещается в системный сетевой буфер, а ее реальная отправка может произойти после возврата из системного вызова. Вызов `sendto()` может блокироваться, если в сетевом буфере не хватает места для датаграммы.

Для чтения принятых датаграмм и определения адреса получателя (при необходимости) служит системный вызов `recvfrom()`. В число параметров этого вызова входят:

- дескриптор сокета, через который принимается датаграмма;
- адрес области памяти, куда следует положить данные, составляющие содержательную часть датаграммы;

- максимальная длина, допустимая для датаграммы. Если количество данных датаграммы превышает заданную максимальную длину, то вызов по умолчанию рассматривает это как ошибочную ситуацию;
- флаги, определяющие поведение системного вызова (в нашем случае они будут полагаться равными 0);
- указатель на структуру, в которую при необходимости может быть занесен адрес сокета отправителя. Если этот адрес не требуется, то можно указать значение NULL;
- указатель на переменную, содержащую максимально возможную длину адреса отправителя. После возвращения из системного вызова в нее будет занесена фактическая длина структуры, содержащей адрес отправителя. Если предыдущий параметр имеет значение NULL, то и этот параметр может иметь значение NULL.

Системный вызов recvfrom() по умолчанию блокируется, если отсутствуют принятые датаграммы, до тех пор, пока датаграмма не появится. При возникновении ошибки он возвращает отрицательное значение, при нормальной работе – длину принятой датаграммы.

Системные вызовы sendto и recvfrom

Прототипы системных вызовов

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sockd, char *buff, int nbytes,
   int flags, struct sockaddr *to, int addrlen);
int recvfrom(int sockd, char *buff, int nbytes,
   int flags, struct sockaddr *from, int *addrlen);
```

Описание системных вызовов

Системный вызов sendto предназначен для отправки датаграмм. Системный вызов recvfrom предназначен для чтения пришедших датаграмм и определения адреса отправителя. По умолчанию при отсутствии пришедших датаграмм вызов recvfrom блокируется до тех пор, пока не появится датаграмма. Вызов sendto может блокироваться при отсутствии места под датаграмму в сетевом буфере. Данное описание не является полным описанием системных вызовов, а предназначено только для использования в нашем курсе. За полной информацией обращайтесь к UNIX Manual.

Параметр sockd является дескриптором созданного ранее сокета, т. е. значением, возвращенным системным вызовом socket(), через который будет отсылаться или получаться информация.

Параметр buff представляет собой адрес области памяти, начиная с которого будет браться информация для передачи или размещаться принятая информация.

Параметр nbytes для системного вызова sendto определяет количество байт, которое должно быть передано, начиная с адреса памяти buff. Параметр nbytes для системного вызова recvfrom определяет максимальное количество байт, которое может быть размещено в приемном буфере, начиная с адреса buff.

Параметр to для системного вызова sendto определяет ссылку на структуру, содержащую адрес сокета получателя информации, которая должна быть заполнена перед вызовом. Если параметр from для системного вызова recvfrom не равен NULL, то для случая установления связи через пакеты данных он определяет ссылку на структуру, в которую будет занесен адрес сокета отправителя информации после завершения вызова. В этом случае перед вызовом эту структуру необходимо обнулить.

Параметр addrlen для системного вызова sendto должен содержать фактическую длину структуры, адрес которой передается в качестве параметра to. Для системного вызова recvfrom параметр addrlen является ссылкой на переменную, в которую будет занесена фактическая длина структуры адреса сокета отправителя, если это определено параметром from. **Заметим, что перед вызовом этот параметр должен указывать на переменную, содержащую максимально допустимое значение такой длины.** Если параметр from имеет значение NULL, то и параметр addrlen может иметь значение NULL.

Параметр flags определяет режимы использования системных вызовов. Рассматривать его применение мы в данном курсе не будем, и поэтому берем значение этого параметра равным 0.

Возвращаемое значение

В случае успешного завершения системный вызов возвращает количество реально отосланных или принятых байт. При возникновении какой-либо ошибки возвращается отрицательное значение.

Определение IP-адресов для вычислительного комплекса

Для определения IP-адресов на компьютере можно воспользоваться утилитой /sbin/ifconfig. Эта утилита выдает всю информацию о сетевых интерфейсах, сконфигурированных в вычислительной системе. Пример выдачи утилиты показан ниже:

```
eth0  Link encap:Ethernet HWaddr 00:90:27:A7:1B:FE
      inet addr:192.168.253.12 Bcast:192.168.253.255
      Mask:255.255.255.0
      UP BROADCAST NOTRAILERS RUNNING MULTICAST MTU:1500
      Metric:1 RX packets:122556059 errors:0 dropped:0
      overruns:0 frame:0 TX packets:116085111 errors:0
      dropped:0 overruns:0 carrier:0 collisions:0
```

```
        txqueuelen:100 RX bytes:2240402748 (2136.6 Mb)
        TX bytes:3057496950 (2915.8 Mb) Interrupt:10
        Base address:0x1000
lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        UP LOOPBACK RUNNING MTU:16436 Metric:1
        RX packets:403 errors:0 dropped:0 overruns:0 frame:0
        TX packets:403 errors:0 dropped:0 overruns:0
        carrier:0 collisions:0 txqueuelen:0
        RX bytes:39932 (38.9 Kb) TX bytes:39932 (38.9 Kb)
```

Сетевой интерфейс `eth0` использует протокол Ethernet. Физический 48-битовый адрес, зашитый в сетевой карте, – `00:90:27:A7:1B:FE`. Его IP-адрес – `192.168.253.12`.

Сетевой интерфейс `lo` не относится ни к какой сетевой карте. Это так называемый локальный интерфейс, который через общую память эмулирует работу сетевой карты для взаимодействия процессов, находящихся на одной машине по полным сетевым адресам. Наличие этого интерфейса позволяет отлаживать сетевые программы на машинах, не имеющих сетевых карт. Его IP-адрес обычно одинаков на всех компьютерах – `127.0.0.1`.

Пример программы UDP-клиента

Рассмотрим, наконец, простой пример программы 14–15-1.с. Эта программа является UDP-клиентом для стандартного системного сервиса `echo`. Стандартный сервис принимает от клиента текстовую датаграмму и, не изменяя ее, отправляет обратно. За сервисом зарезервирован номер порта 7. Для правильного запуска программы необходимо указать символьный IP-адрес сетевого интерфейса компьютера, к сервису которого нужно обратиться, в качестве аргумента командной строки, например:

```
a.out 192.168.253.12
```

Ниже следует текст программы:

```
/* Простой пример UDP клиента для сервиса echo */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
```

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
int main(int argc, char **argv)
{
  int sockfd; /* Дескриптор сокета */
  int n, len; /* Переменные для различных длин и
    количества символов */
  char sendline[1000], recvline[1000]; /* Массивы
    для отсылаемой и принятой строки */
  struct sockaddr_in servaddr, cliaddr; /* Структуры
  для адресов сервера и клиента */
  /* Сначала проверяем наличие второго аргумента в
  командной строке. При его отсутствии ругаемся и
  прекращаем работу */
  if(argc != 2){
    printf("Usage: a.out <IP address>\n");
    exit(1);
  }
  /* Создаем UDP сокет */
  if((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0){
    perror(NULL); /* Печатаем сообщение об ошибке */
    exit(1);
  }
  /* Заполняем структуру для адреса клиента: семейство
  протоколов TCP/IP, сетевой интерфейс - любой, номер
  порта по усмотрению операционной системы. Поскольку
  в структуре содержится дополнительное не нужное нам
  поле, которое должно быть нулевым, перед заполнением
  обнуляем ее всю */
  bzero(&cliaddr, sizeof(cliaddr));
  cliaddr.sin_family = AF_INET;
  cliaddr.sin_port = htons(0);
  cliaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  /* Настраиваем адрес сокета */
  if(bind(sockfd, (struct sockaddr *) &cliaddr,
  sizeof(cliaddr)) < 0){
    perror(NULL);
    close(sockfd); /* По окончании работы закрываем
    дескриптор сокета */
    exit(1);
  }
```

```
    /* Заполняем структуру для адреса сервера:
    семейство протоколов TCP/IP, сетевой интерфейс -
    из аргумента командной строки, номер порта 7.
    Поскольку в структуре содержится дополнительное
    не нужное нам поле, которое должно быть нулевым,
    перед заполнением обнуляем ее всю */
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(7);
    if(inet_aton(argv[1], &servaddr.sin_addr) == 0){
      printf("Invalid IP address\n");
      close(sockfd); /* По окончании работы закрываем
        дескриптор сокета */
      exit(1);
    }
    /* Вводим строку, которую отошлем серверу */
    printf("String => ");
    fgets(sendline, 1000, stdin);
    /* Отсылаем датаграмму */
    if(sendto(sockfd, sendline, strlen(sendline)+1,
    0, (struct sockaddr *) &servaddr,
    sizeof(servaddr)) < 0){
    perror(NULL);
    close(sockfd);
    exit(1);
    }
    /* Ожидаем ответа и читаем его. Максимальная
    допустимая длина датаграммы - 1000 символов,
    адрес отправителя нам не нужен */
    if((n = recvfrom(sockfd, recvline, 1000, 0,
    (struct sockaddr *) NULL, NULL)) < 0){
      perror(NULL);
      close(sockfd);
      exit(1);
    }
    /* Печатаем пришедший ответ и закрываем сокет */
    printf("%s\n", recvline);
    close(sockfd);
    return 0;
}
```

Наберите и откомпилируйте программу. Перед запуском **«узнайте у своего системного администратора»**, запущен ли в системе стандартный UDP-сервис echo и если нет, попросите стартовать его. Запустите программу с запросом к сервису своего компьютера, к сервисам других компьютеров. Если в качестве IP-адреса указать несуществующий адрес, адрес выключенной машины или машины, на которой не работает сервис echo, то программа бесконечно блокируется в вызове recvfrom(), ожидая ответа. Протокол UDP не является надежным протоколом. Если датаграмму доставить по назначению не удалось, то отправитель никогда об этом не узнает!

Пример программы UDP-сервера

Поскольку UDP-сервер использует те же самые системные вызовы, что и UDP-клиент, мы можем сразу приступить к рассмотрению примера UDP-сервера (программа 15–16-2.c) для сервиса echo.

```
/* Простой пример UDP-сервера для сервиса echo */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
int main()
{
   int sockfd; /* Дескриптор сокета */
   int clilen, n; /* Переменные для различных длин
      и количества символов */
   char line[1000]; /* Массив для принятой и
      отсылаемой строки */
   struct sockaddr_in servaddr, cliaddr; /* Структуры
      для адресов сервера и клиента */
   /* Заполняем структуру для адреса сервера: семейство
   протоколов TCP/IP, сетевой интерфейс - любой, номер
   порта 51000. Поскольку в структуре содержится
   дополнительное не нужное нам поле, которое должно
   быть нулевым, перед заполнением обнуляем ее всю */
   bzero(&servaddr, sizeof(servaddr));
   servaddr.sin_family = AF_INET;
   servaddr.sin_port = htons(51000);
```

```
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    /* Создаем UDP-сокет */
    if((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0){
      perror(NULL); /* Печатаем сообщение об ошибке */
      exit(1);
    }
    /* Настраиваем адрес сокета */
    if(bind(sockfd, (struct sockaddr *) &servaddr,
    sizeof(servaddr)) < 0){
      perror(NULL);
      close(sockfd);
      exit(1);
    }
    while(1) {
      /* Основной цикл обслуживания*/
      /* В переменную clilen заносим максимальную
      длину для ожидаемого адреса клиента */
      clilen = sizeof(cliaddr);
      /* Ожидаем прихода запроса от клиента и читаем
      его. Максимальная допустимая длина датаграммы -
      999 символов, адрес отправителя помещаем в
      структуру cliaddr, его реальная длина будет
      занесена в переменную clilen */
      if((n = recvfrom(sockfd, line, 999, 0,
      (struct sockaddr *) &cliaddr, &clilen)) < 0){
        perror(NULL);
        close(sockfd);
        exit(1);
      }
      /* Печатаем принятый текст на экране */
      printf("%s\n", line);
      /* Принятый текст отправляем обратно по адресу
      отправителя */
      if(sendto(sockfd, line, strlen(line), 0,
      (struct sockaddr *) &cliaddr, clilen) < 0){
        perror(NULL);
        close(sockfd);
        exit(1);
      } /* Уходим ожидать новую датаграмму*/
    }
    return 0;
}
```

Наберите и откомпилируйте программу. Запустите ее на выполнение. Модифицируйте текст программы UDP-клиента (программа 14–15-1.c), заменив номер порта с 7 на 51000. Запустите клиента с другого виртуального терминала или с другого компьютера и убедитесь, что клиент и сервер взаимодействуют корректно.

Организация связи между процессами с помощью установки логического соединения

Теперь посмотрим, какие действия нам понадобятся для организации взаимодействия процессов с помощью протокола TCP, то есть при помощи создания логического соединения. И начнем, как и в разделе «Использование модели клиент — сервер для взаимодействия удаленных процессов» текущего семинара, с простой жизненной аналогии. Если взаимодействие процессов через датаграммы напоминает общение людей по переписке, то для протокола TCP лучшей аналогией является общение людей по телефону.

Какие действия должен выполнить клиент для того, чтобы связаться по телефону с сервером? Во-первых, необходимо приобрести телефон (создать сокет), во-вторых, подключить его на АТС – получить номер (настроить адрес сокета). Далее требуется позвонить серверу (установить логическое соединение). После установления соединения можно неоднократно обмениваться с сервером информацией (писать и читать из потока данных). По окончании взаимодействия нужно повесить трубку (закрыть сокет).

Первые действия сервера аналогичны действиям клиента. Он должен приобрести телефон и подключить его на АТС (создать сокет и настроить его адрес). А вот дальше поведение клиента и сервера различно. Представьте себе, что телефоны изначально продаются с выключенным звонком. Звонить по ним можно, а вот принять звонок – нет. Для того чтобы вы могли пообщаться, необходимо включить звонок. В терминах сокетов это означает, что TCP-сокет по умолчанию создается в активном состоянии и предназначен не для приема, а для установления соединения. Для того чтобы соединение принять, сокет требуется перевести в пассивное состояние.

Если два человека беседуют по телефону, то попытка других людей дозвониться до них окажется неудачной. Будет идти сигнал «занято», и соединение не установится. В то же время хотелось бы, чтобы клиент в такой ситуации не получал отказ в обслуживании, а ожидал своей очереди. Подобное наблюдается в различных телефонных справочных, когда вы слышите «Ждите, пожалуйста, ответа. Вам обязательно ответит оператор». Поэтому следующее действие сервера – это создание очереди для

обслуживания клиентов. Далее сервер должен дождаться установления соединения, прочитать информацию, переданную по линии связи, обработать ее и отправить полученный результат обратно. Обмен информацией может осуществляться неоднократно. Заметим, что сокет, находящийся в пассивном состоянии, не предназначен для операций приема и передачи информации. Для общения на сервере во время установления соединения автоматически создается новый потоковый сокет, через который и производится обмен данными с клиентами. По окончании общения сервер «кладет трубку» (закрывает этот новый сокет) и отправляется ждать очередного звонка.

Схематично эти действия выглядят так, как показано на рисунке 14-15.7. Как и в случае протокола UDP отдельным действиям или их группам соответствуют системные вызовы, частично совпадающие с вызовами для протокола UDP. Их названия написаны справа от блоков соответствующих действий.

Для протокола TCP неравноправность процессов клиента и сервера видна особенно отчетливо в различии используемых системных вызовов. Для создания сокетов и там, и там по-прежнему используется системный вызов `socket()`. Затем наборы системных вызовов становятся различными.

Для привязки сервера к IP-адресу и номеру порта, как и в случае UDP-протокола, используется системный вызов `bind()`. Для процесса клиента эта привязка объединена с процессом установления соединения с сервером в новом системном вызове `connect()` и скрыта от глаз пользователя. Внутри этого вызова операционная система осуществляет настройку сокета на выбранный ею порт и на адрес любого сетевого интерфейса. Для перевода сокета на сервере в пассивное состояние и для создания очереди соединений служит системный вызов `listen()`. Сервер ожидает соединения и получает информацию об адресе соединившегося с ним клиента с помощью системного вызова `accept()`. Поскольку установленное логическое соединение выглядит со стороны процессов как канал связи, позволяющий обмениваться данными с помощью потоковой модели, для передачи и чтения информации оба системных вызова используют уже известные нам системные вызовы `read()` и `write()`, а для завершения соединения – системный вызов `close()`. Необходимо отметить, что при работе с сокетами вызовы `read()` и `write()` обладают теми же особенностями поведения, что и при работе с pip'ами и FIFO (см. семинар 5).

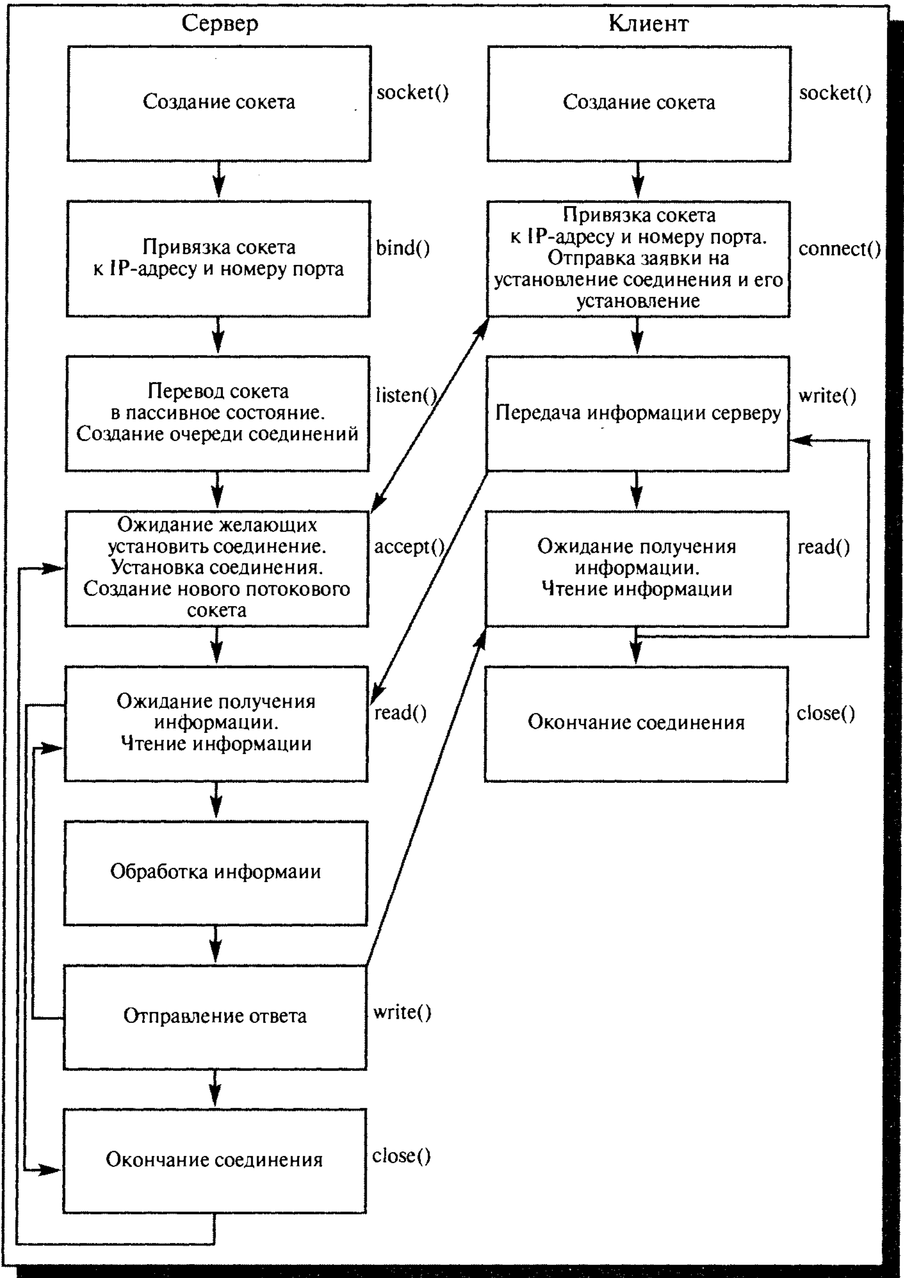


Рис. 14-15.7. Схема взаимодействия клиента и сервера для протокола TCP

Установление логического соединения. Системный вызов connect()

Среди системных вызовов со стороны клиента появляется только один новый – `connect()`. Системный вызов `connect()` при работе с TCP-сокетами служит для установления логического соединения со стороны клиента. Вызов `connect()` скрывает внутри себя настройку сокета на выбранный системой порт и произвольный сетевой интерфейс (по сути дела, вызов `bind()` с нулевым номером порта и IP-адресом `INADDR_ANY`). Вызов блокируется до тех пор, пока не будет установлено логическое соединение, или пока не пройдет определенный промежуток времени, который может регулироваться системным администратором.

Для установления соединения необходимо задать три параметра: дескриптор активного сокета, через который будет устанавливаться соединение, полный адрес сокета сервера и его длину.

Системный вызов connect()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockd, struct sockaddr *servaddr, int addrlen);
```

Описание системного вызова

Системный вызов `connect` служит для организации связи клиента с сервером. Чаще всего он используется для установления логического соединения, хотя может быть применен и при связи с помощью датаграмм (connectionless). Данное описание не является полным описанием системного вызова, а предназначено только для использования в нашем курсе. Полную информацию можно найти в UNIX Manual.

Параметр `sockd` является дескриптором созданного ранее коммуникационного узла, т. е. значением, которое вернул системный вызов `socket()`.

Параметр `servaddr` представляет собой адрес структуры, содержащей информацию о полном адресе сокета сервера. Он имеет тип указателя на структуру-шаблон `struct sockaddr`, которая должна быть конкретизирована в зависимости от используемого семейства протоколов и заполнена перед вызовом.

Параметр `addrlen` должен содержать фактическую длину структуры, адрес которой передается в качестве второго параметра. Эта длина меняется в зависимости от семейства протоколов и различается даже в пределах одного семейства протоколов (например, для UNIX Domain).

При установлении виртуального соединения системный вызов не возвращается до его установления или до истечения установленного в системе времени – timeout. При использовании его в connectionless связи вызов возвращается немедленно.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и отрицательное значение, если в процессе его выполнения возникла ошибка.

Пример программы TCP-клиента

Рассмотрим пример – программу 14–15-3.с. Это простой TCP-клиент, обращающийся к стандартному системному сервису echo. Стандартный сервис принимает от клиента текстовую датаграмму и, не изменяя ее, отправляет обратно. За сервисом зарезервирован номер порта 7. Заметим, что это порт 7 TCP – не путать с портом 7 UDP из примера в разделе «Пример программы UDP-клиента»! Для правильного запуска программы необходимо указать символьный IP-адрес сетевого интерфейса компьютера, к сервису которого требуется обратиться, в качестве аргумента командной строки, например:

```
a.out 192.168.253.12
```

Для того чтобы подчеркнуть, что после установления логического соединения клиент и сервер могут обмениваться информацией неоднократно, клиент трижды запрашивает текст с экрана, отсылает его серверу и печатает полученный ответ. Ниже представлен текст программы.

```
/* Простой пример TCP-клиента для сервиса echo */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
void main(int argc, char **argv)
{
  int sockfd; /* Дескриптор сокета */
  int n; /* Количество переданных или прочитанных
    символов */
  int i; /* Счетчик цикла */
  char sendline[1000],recvline[1000]; /* Массивы
    для отсылаемой и принятой строки */
```

```
struct sockaddr_in servaddr; /* Структура для
  адреса сервера */
/* Сначала проверяем наличие второго аргумента в
командной строке. При его отсутствии прекращаем
работу */
if(argc != 2){
  printf("Usage: a.out <IP address>\n");
  exit(1);
}
/* Обнуляем символьные массивы */
bzero(sendline,1000);
bzero(recvline,1000);
/* Создаем TCP-сокет */
if((sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0){
  perror(NULL); /* Печатаем сообщение об ошибке */
  exit(1);
}
/* Заполняем структуру для адреса сервера: семейство
протоколов TCP/IP, сетевой интерфейс - из аргумента
командной строки, номер порта 7. Поскольку в структуре
содержится дополнительное не нужное нам поле, которое
должно быть нулевым, перед заполнением обнуляем ее всю */
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(51000);
if(inet_aton(argv[1], &servaddr.sin_addr) == 0){
  printf("Invalid IP address\n");
  close(sockfd);
  exit(1);
}
/* Устанавливаем логическое соединение через
созданный сокет с сокетом сервера, адрес которого
мы занесли в структуру */
if(connect(sockfd, (struct sockaddr *) &servaddr,
sizeof(servaddr)) < 0){
  perror(NULL);
  close(sockfd);
  exit(1);
}
/* Три раза в цикле вводим строку с клавиатуры,
отправляем ее серверу и читаем полученный ответ */
for(i=0; i<3; i++){
```

```
        printf("String => ");
        fflush(stdin);
        fgets(sendline, 1000, stdin);
        if( (n = write(sockfd, sendline,
        strlen(sendline)+1)) < 0){
            perror("Can\'t write\n");
            close(sockfd);
            exit(1);
        }
        if ( (n = read(sockfd,recvline, 999)) < 0){
            perror("Can\'t read\n");
            close(sockfd);
            exit(1);
        }
        printf("%s", recvline);
    }
    /* Завершаем соединение */
    close(sockfd);
}
```

Наберите и откомпилируйте программу. Перед запуском **узнайте у своего системного администратора**, запущен ли в системе стандартный TCP-сервис `echo` и, если нет, попросите это сделать. Запустите программу с запросом к сервису своего компьютера, к сервисам других компьютеров. Если в качестве IP-адреса указать несуществующий адрес или адрес выключенной машины, то программа сообщит об ошибке при работе вызова `connect()` (правда, возможно, придется подождать окончания time-out'а). При задании адреса компьютера, на котором не работает сервис `echo`, об ошибке станет известно сразу же. Протокол TCP является надежным протоколом. Если логическое соединение установить не удалось, то отправитель будет знать об этом.

Как происходит установление виртуального соединения

Протокол TCP является надежным дуплексным протоколом. С точки зрения пользователя работа через протокол TCP выглядит как обмен информацией через поток данных. Внутри сетевых частей операционных систем поток данных отправителя нарезается на пакеты данных, которые, собственно, путешествуют по сети и на машине-получателе вновь собираются в выходной поток данных. В лекции 4 речь шла о том, каким образом может обеспечиваться надежность передачи информации в средствах связи,

использующих в своей основе передачу пакетов данных. В протоколе TCP используются приемы нумерации передаваемых пакетов и контроля порядка их получения, подтверждения о приеме пакета со стороны получателя и насчет контрольных сумм по передаваемой информации. Для правильного порядка получения пакетов получатель должен знать начальный номер первого пакета отправителя. Поскольку связь является дуплексной, и в роли отправителя пакетов данных могут выступать обе взаимодействующие стороны, они до передачи пакетов данных должны обменяться, по крайней мере, информацией об их начальных номерах. Согласование начальных номеров происходит по инициативе клиента при выполнении системного вызова `connect()`.

Для такого согласования клиент посылает серверу специальный пакет информации, который принято называть SYN (от слова **syn**chronize – синхронизировать). Он содержит, как минимум, начальный номер для пакетов данных, который будет использовать клиент. Сервер должен подтвердить получение пакета SYN от клиента и отправить ему свой пакет SYN с начальным номером для пакетов данных, в виде единого пакета с сегментами SYN и ACK (от слова **ack**nowledgement – подтверждение). В ответ клиент пакетом данных ACK должен подтвердить прием пакета данных от сервера.

Описанная выше процедура, получившая название *трехэтапного рукопожатия* (*three-way handshake*), схематично изображена на рисунке 14-15.8. При приеме на машине-сервере пакета SYN, направленного на пассивный (слушающий) сокет, сетевая часть создает операционной системе копию этого сокета – присоединенный сокет – для последующего общения, отмечая его как сокет с не полностью установленным соединением. После приема от клиента пакета ACK этот сокет переводится в состояние полностью установленного соединения, и тогда он готов к дальнейшей работе с использованием вызовов `read()` и `write()`.

Системный вызов listen()

Системный вызов `listen()` является первым из еще неизвестных нам вызовов, применяемым на TCP-сервере. В его задачу входит перевод TCP-сокета в пассивное (слушающее) состояние и создание очередей для порождаемых при установлении соединения присоединенных сокетов, находящихся в состоянии не полностью установленного соединения и полностью установленного соединения. Для этого вызов имеет два параметра: дескриптор TCP-сокета и число, определяющее глубину создаваемых очередей.

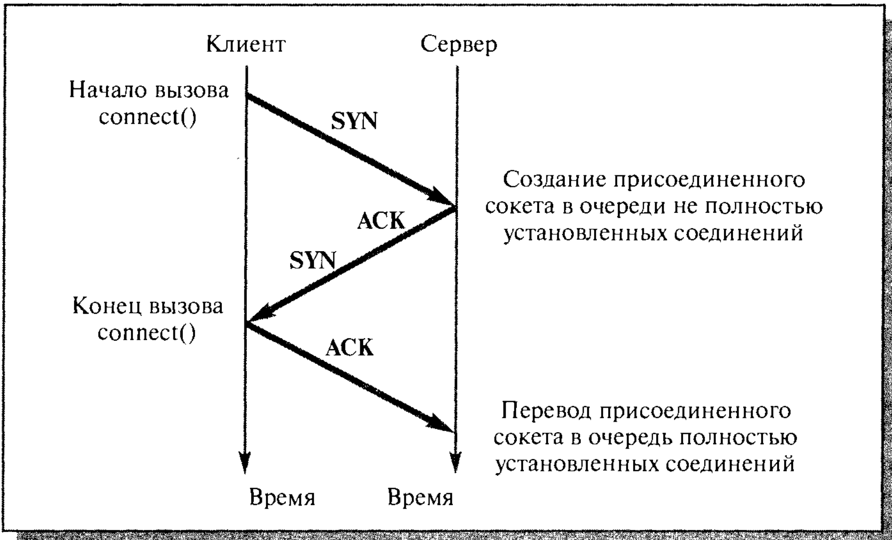


Рис. 14-15.8. Схема установления TCP-соединения

Системный вызов listen()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sockd, int backlog);
```

Описание системного вызова

Системный вызов `listen` используется сервером, ориентированным на установление связи путем виртуального соединения, для перевода сокета в пассивный режим и установления глубины очереди для соединений.

Параметр `sockd` является дескриптором созданного ранее сокета, который должен быть переведен в пассивный режим, т. е. значением, которое вернул системный вызов `socket()`. Системный вызов `listen` требует предварительной настройки адреса сокета с помощью системного вызова `bind()`.

Параметр `backlog` определяет максимальный размер очередей для сокетов, находящихся в состояниях полностью и не полностью установленных соединений.

Возвращаемое значение

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

Последний параметр на разных UNIX-подобных операционных системах и даже на разных версиях одной и той же системы может иметь различный смысл. Где-то это суммарная длина обеих очередей, где-то он относится к очереди не полностью установленных соединений (например, Linux до версии ядра 2.2) где-то – к очереди полностью установленных соединений (например, Linux, начиная с версии ядра 2.2), где-то – вообще игнорируется.

Системный вызов accept()

Системный вызов `accept()` позволяет серверу получить информацию о полностью установленных соединениях. Если очередь полностью установленных соединений не пуста, то он возвращает дескриптор для первого присоединенного сокета в этой очереди, одновременно удаляя его из очереди. Если очередь пуста, то вызов ожидает появления полностью установленного соединения. Системный вызов также позволяет серверу узнать полный адрес клиента, установившего соединение. У вызова есть три параметра: дескриптор слушающего сокета, через который ожидается установление соединения; указатель на структуру, в которую при необходимости будет занесен полный адрес сокета клиента, установившего соединение; указатель на целую переменную, содержащую максимально допустимую длину этого адреса. Как и в случае вызова `recvfrom()`, последний параметр является модернизируемым, а если нас не интересует, кто с нами соединился, то вместо второго и третьего параметров можно указать значение `NULL`.

Системный вызов accept()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockd, struct sockaddr *cliaddr,
   int *clilen);
```

Описание системного вызова

Системный вызов `accept` используется сервером, ориентированным на установление связи путем виртуального соединения, для приема полностью установленного соединения.

Параметр `sockd` является дескриптором созданного и настроенного сокета, предварительного переведенного в пассивный (слушающий) режим с помощью системного вызова `listen()`.

Системный вызов `accept` требует предварительной настройки адреса сокета с помощью системного вызова `bind()`.

Параметр `cliaddr` служит для получения адреса клиента, установившего логическое соединение, и должен содержать указатель на структуру, в которую будет занесен этот адрес.

Параметр `clilen` содержит указатель на целую переменную, которая после возвращения из вызова будет содержать фактическую длину адреса клиента. **Заметим, что перед вызовом эта переменная должна содержать** максимально допустимое значение такой длины. Если параметр `cliaddr` имеет значение `NULL`, то и параметр `clilen` может иметь значение `NULL`.

Возвращаемое значение

Системный вызов возвращает при нормальном завершении дескриптор присоединенного сокета, созданного при установлении соединения для последующего общения клиента и сервера, и значение `-1` при возникновении ошибки.

Пример простого TCP-сервера

Рассмотрим программу 14—15-4.с, реализующую простой TCP-сервер для сервиса `echo`:

```
/* Пример простого TCP-сервера для сервиса echo */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
void main()
{
  int sockfd, newsockfd; /* Дескрипторы для
  слушающего и присоединенного сокетов */
  int clilen; /* Длина адреса клиента */
  int n; /* Количество принятых символов */
  char line[1000]; /* Буфер для приема информации */
  struct sockaddr_in servaddr, cliaddr; /* Структуры
    для размещения полных адресов сервера и
    клиента */
  /* Создаем TCP-сокет */
  if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
    perror(NULL);
    exit(1);
  }
```

```
/* Заполняем структуру для адреса сервера: семейство
протоколов TCP/IP, сетевой интерфейс - любой, номер
порта 51000. Поскольку в структуре содержится
дополнительное не нужное нам поле, которое должно
быть нулевым, обнуляем ее всю перед заполнением */
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family= AF_INET;
servaddr.sin_port= htons(51000);
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
/* Настраиваем адрес сокета */
if(bind(sockfd, (struct sockaddr *) &servaddr,
sizeof(servaddr)) < 0){
  perror(NULL);
  close(sockfd);
  exit(1);
}
/* Переводим созданный сокет в пассивное (слушающее)
состояние. Глубину очереди для установленных
соединений описываем значением 5 */
if(listen(sockfd, 5) < 0){
  perror(NULL);
  close(sockfd);
  exit(1);
}
/* Основной цикл сервера */
while(1){
  /* В переменную clilen заносим максимальную
  длину ожидаемого адреса клиента */
  clilen = sizeof(cliaddr);
  /* Ожидаем полностью установленного соединения
  на слушающем сокете. При нормальном завершении
  у нас в структуре cliaddr будет лежать полный
  адрес клиента, установившего соединение, а в
  переменной clilen - его фактическая длина. Вызов
  же вернет дескриптор присоединенного сокета,
  через который будет происходить общение с
  клиентом. Заметим, что информация о клиенте у
  нас в дальнейшем никак не используется, поэтому
  вместо второго и третьего параметров можно было
  поставить значения NULL. */
  if((newsockfd = accept(sockfd,
  (struct sockaddr *) &cliaddr, &clilen)) < 0){
```

```
            perror(NULL);
            close(sockfd);
            exit(1);
        }
        /* В цикле принимаем информацию от клиента до
        тех пор, пока не произойдет ошибки (вызов read()
        вернет отрицательное значение) или клиент не
        закроет соединение (вызов read() вернет значение
        0). Максимальную длину одной порции данных от
        клиента ограничим 999 символами. В операциях
        чтения и записи пользуемся дескриптором
        присоединенного сокета, т. е. значением, которое
        вернул вызов accept().*/
        while((n = read(newsockfd, line, 999)) > 0){
            /* Принятые данные отправляем обратно */
            if((n = write(newsockfd, line,
            strlen(line)+1)) < 0){
                perror(NULL);
                close(sockfd);
                close(newsockfd);
                exit(1);
            }
        }
    /* Если при чтении возникла ошибка - завершаем
    работу */
    if(n < 0){
            perror(NULL);
            close(sockfd);
            close(newsockfd);
            exit(1);
        }
        /* Закрываем дескриптор присоединенного сокета
        и уходим ожидать нового соединения */
        close(newsockfd);
    }
}
```

Наберите и откомпилируйте программу. Запустите ее на выполнение. Модифицируйте текст программы TCP-клиента (программа 14–15-3.c), заменив номер порта с 7 на 51000. Запустите клиента с другого виртуального терминала или с другого компьютера и убедитесь, что клиент и сервер взаимодействуют корректно.

Создание программы с параллельной обработкой запросов клиентов

В приведенном выше примере сервер осуществлял последовательную обработку запросов от разных клиентов. При таком подходе клиенты могут подолгу простаивать после установления соединения, ожидая обслуживания. Поэтому обычно применяется схема псевдопараллельной обработки запросов. После приема установленного соединения сервер порождает процесс-ребенок, которому и поручает дальнейшую работу с клиентом. Процесс-родитель закрывает присоединенный сокет и уходит на ожидание нового соединения. Схематично организация такого сервера изображена на рис. 14-15.9.

Напишите, откомпилируйте и запустите такой параллельный сервер. Убедитесь в его работоспособности. Не забудьте о необходимости удаления зомби-процессов.

Применение интерфейса сетевых вызовов для других семейств протоколов. UNIX Domain протоколы. Файлы типа «сокет»

Рассмотренный нами интерфейс умеет работать не только со стеком протоколов TCP/IP, но и с другими семействами протоколов. При этом требуется лишь незначительное изменение написанных с его помощью программ. Рассмотрим действия, которые необходимо выполнить для модернизации написанных для TCP/IP программ под другое семейство протоколов:

1. Изменяется тип сокета, поэтому для его точной спецификации нужно задавать другие параметры в системном вызове `socket()`.
2. В различных семействах протоколов применяются различные адресные пространства для удаленных и локальных адресов сокетов. Поэтому меняется состав структуры для хранения полного адреса сокета, название ее типа, наименования полей и способ их заполнения.
3. Описание типов данных и предопределенных констант будет находиться в других include-файлах, поэтому потребуется заменить include-файлы `<netinet/in.h>` и `<arpa/inet.h>` на файлы, относящиеся к выбранному семейству протоколов.
4. Может измениться способ вычисления фактической длины полного адреса сокета и указания его максимального размера.

И все!!!

Давайте подробнее рассмотрим эти изменения на примере семейства UNIX Domain протоколов. Семейство UNIX Domain протоколов предназначено для общения локальных процессов с использованием интер-

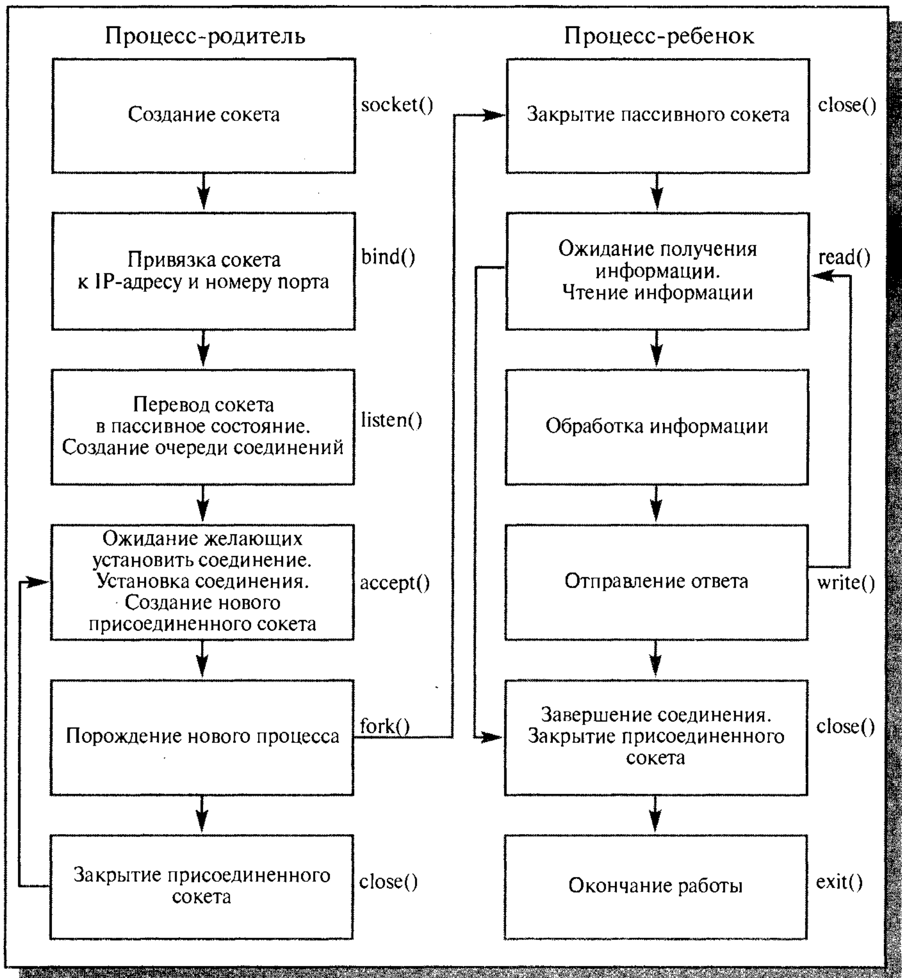


Рис. 14-15.9. Схема работы TCP-сервера с параллельной обработкой запросов

фейса системных вызовов. Оно содержит один потоковый и один дата-граммный протокол. Никакой сетевой интерфейс при этом не используется, а вся передача информации реально происходит через адресное пространство ядра операционной системы. Многие программы, взаимодействующие и с локальными, и с удаленными процессами (например, X-Windows), для локального общения используют этот стек протоколов.

Поскольку общение происходит в рамках одной вычислительной системы, в полном адресе сокета его удаленная часть отсутствует. В качестве

адресного пространства портов – локальной части адреса – выбрано адресное пространство, совпадающее с множеством всех допустимых имен файлов в файловой системе.

При этом в качестве имени сокета требуется задавать имя несуществующего еще файла в директории, к которой у вас есть права доступа как на запись, так и на чтение. При настройке адреса (системный вызов bind()) под этим именем будет создан файл типа «сокет» – последний еще неизвестный нам тип файла. Этот файл для сокетов играет роль файла-метки типа FIFO для именованных pip'ов. Если на вашей машине функционируют X-Windows, то вы сможете обнаружить такой файл в директории с именем /tmp/.X11-unix – это файл типа «сокет», служащий для взаимодействия локальных процессов с оконным сервером.

Для хранения полного адреса сокета используется структура следующего вида, описанного в файле <sys/un.h>:

```
struct sockaddr _un{
  short sun_family; /* Избранное семейство
    протоколов - всегда AF_UNIX */
  char sun_path[108]; /* Имя файла типа "сокет" */
};
```

Выбранное имя файла мы будем копировать внутрь структуры, используя функцию strcpy().

Фактическая длина полного адреса сокета, хранящегося в структуре с именем my_addr, может быть вычислена следующим образом: sizeof(short)+strlen(my_addr.sun_path). В Linux для этих целей можно использовать специальный макрос языка C:

```
SUN_LEN(struct sokaddr_un*)
```

Ниже приведены тексты переписанных под семейство UNIX Domain протоколов клиента и сервера для сервиса echo (программы 14–15-5.c и 14–15-6.c), общающиеся через датаграммы. Клиент использует сокет с именем AAAA в текущей директории, а сервер – сокет с именем BBBB. Как следует из описания типа данных, эти имена (полные или относительные) не должны по длине превышать 107 символов. Комментарии даны лишь для изменений по сравнению с программами 14–15-1.c и 14–15-2.c.

```
/* A simple echo UNIX Domain datagramm server */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h> /* Новый include-файл вместо
```

```
  netinet/in.h и arpa/inet.h */
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
int main()
{
  int sockfd;
  int clilen, n;
  char line[1000];
  struct sockaddr_un servaddr, cliaddr; /* новый
    тип данных под адреса сокетов */
  if((sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
  /* Изменен тип семейства протоколов */
  {
    perror(NULL);
    exit(1);
  }
  bzero(&servaddr, sizeof(servaddr));
  servaddr.sun_family = AF_UNIX; /* Изменен тип
    семейства протоколов и имя поля в структуре */
  strcpy(servaddr.sun_path,"BBBB"); /* Локальный
    адрес сокета сервера - BBBB - в текущей директории */
  if(bind(sockfd, (struct sockaddr *) &servaddr,
  SUN_LEN(&servaddr)) < 0) /* Изменено вычисление
    фактической длины адреса */
  {
    perror(NULL);
    close(sockfd);
    exit(1);
  }
  while(1) {
    clilen = sizeof(struct sockaddr_un); /* Изменено
      вычисление максимальной длины для адреса клиента */
    if((n = recvfrom(sockfd, line, 999, 0,
    (struct sockaddr *) &cliaddr, &clilen)) < 0){
      perror(NULL);
      close(sockfd);
      exit(1);
    }
    if(sendto(sockfd, line, strlen(line), 0,
    (struct sockaddr *) &cliaddr, clilen) < 0){
```

```
        perror(NULL);
        close(sockfd);
        exit(1);
      }
    }
    return 0;
  }

  /* A simple echo UNIX Domain datagram client */
  #include <sys/types.h>
  #include <sys/socket.h>
  #include <sys/un.h> /* Новый include-файл вместо
    netinet/in.h и arpa/inet.h */
  #include <string.h>
  #include <stdio.h>
  #include <errno.h>
  #include <unistd.h>
  int main() /* Аргументы командной строки не нужны,
  так как сервис является локальным, и не нужно
    указывать, к какой машине мы обращаемся с запросом */
  {
    int sockfd;
    int n, len;
    char sendline[1000], recvline[1000];
    struct sockaddr_un servaddr, cliaddr; /* новый
    тип данных под адреса сокетов */
    if((sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
    /* Изменен тип семейства протоколов */
    {
      perror(NULL);
      exit(1);
    }
    bzero(&cliaddr, sizeof(cliaddr));
    cliaddr.sun_family= AF_UNIX; /* Изменен тип
      семейства протоколов и имя поля в структуре */
    strcpy(cliaddr.sun_path,"AAAA");/* Локальный адрес
      сокета клиента - AAAA - в текущей директории */
    if(bind(sockfd, (struct sockaddr *) &cliaddr,
    SUN_LEN(&cliaddr)) < 0) /* Изменено вычисление
      фактической длины адреса */
    {
      perror(NULL);
```

```
        close(sockfd);
        exit(1);
    }
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sun_family = AF_UNIX; /* Изменен тип
        семейства протоколов и имя поля в структуре */
    strcpy(servaddr.sun_path,"BBBB"); /* Локальный адрес
        сокета сервера - BBBB - в текущей директории */
    printf("String => ");
    fgets(sendline, 1000, stdin);
    if(sendto(sockfd, sendline, strlen(sendline)+1,
    0, (struct sockaddr *) &servaddr,
    SUN_LEN(&servaddr)) < 0) /* Изменено вычисление
        фактической длины адреса */
    {
        perror(NULL);
        close(sockfd);
        exit(1);
    }
    if((n = recvfrom(sockfd, recvline, 1000, 0,
    (struct sockaddr *) NULL, NULL)) < 0){
        perror(NULL);
        close(sockfd);
        exit(1);
    }
    recvline[n] = 0;
    printf("%s", recvline);
    close(sockfd);
    return 0;
}
```

Наберите программы, откомпилируйте их и убедитесь в их работоспособности.

Создание потоковых клиента и сервера для стека UNIX Domain протоколов

По аналогии с программами в предыдущем примере модифицируйте тексты программ TCP клиента и сервера для сервиса echo (программы 14–15-3.с и 14–15-4.с) для потокового общения в семействе UNIX Domain протоколов. Откомпилируйте их и убедитесь в их правильном функционировании.

Литература

[Bach, 1986] Bach M.J. The design of the UNIX Operating System.— Prentice-Hall, 1986.

[CCITT, 1991] Security Architecture for Open Systems Interconnection for CCITT Applications. Recommendations X.800. CCITT. – Geneva. 1991.

[Denning, 1996] Peter J. Denning. Before memory was virtual, Draft, June 6th 1996, at http://cne.gmu.edu/pjd/PUBS/bvm.pdf.

[DoD, 1993] Department of Defense. Trusted Computer System Evaluation Criteria. – DoD 5200.28, STD. 1993.

[DTI, 1991] Department of Trade and Industry. Information Technology Security Evaluation Criteria (ITSEC). Harmonized Criteria of France – Germany – the Netherlands – the United Kingdom. – London. 1991.

[Intel, 1989] i486™ Microprocessor, Intel Corporation, 1989.

[Linnaes,1789] Linnaeus. Karl, Systema naturae, 13th ed., t. 1-3 – Lugduni, 1789-96

[Ritchie,1984] Ritchie D.M., The Evolution of the Unix Time-sharing System. // AT&T Bell Laboratories Technical Journal 63 No. 6, Part 2, October 1984 – pp. 1577-93.

[Silberschatz, 2002] Silberschatz A., P.B.Galvin, Operating System Concepts, 6th edition. – John Willey & Sons, 2002.

[Stevens, 1990] Stevens R. W, Unix Network Programming – Prentice Hall, Inc., 1990, First edition.

[Ахо, 2001] Ахо В., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы. – М.: Вильямс, 2001.

[Баурн, 1986] Баурн С. Операционная система UNIX. – М.: Мир. 1986.

[Беляков, 1991] Беляков М.И., Рабовер Ю.И., Фридман А.Л. Мобильная операционная система. – М.:, Радио и связь, 1991.

[Блэк, 2001] Блэк У. Интернет: протоколы безопасности. Учебный курс. – Спб.: Издательский дом Питер, 2001.

[Брамм, 1990] Брамм П., Брамм Д. Микропроцессор 80386 и его применение. – М., Мир, 1990.

[Вахалия, 2003] Вахалия Ю. UNIX изнутри. – Спб.: Издательский дом Питер, 2003.

[Дейтел, 1987] Дейтел Г. Введение в операционные системы. – М.: Мир, 1987.

[Дунаев, 1996] Дунаев С. Unix. System V. Release 4.2. – М.: Диалог МИФИ, 1996.

[Казаринов, 1990] Казаринов Ю.М., Номоконов В.М., Подклетнов Г.С., Филиппов Ф.М. Микропроцессорный комплекс К1810. – М.: Высшая школа, 1990.

[Кастер, 1996] Кастер Хелен. Основы Windows NT и NTFS. – М.: Русская редакция. 1996.

[Керниган, 1992] Керниган Б. В., Пайк Р. UNIX – универсальная среда программирования. – М.: Финансы и статистика. 1992.

[Коффрон, 1983] Коффрон Дж. Технические средства микропроцессорных систем. – М.: Мир, 1983.

[Кузнецов] Кузнецов С.Д. Операционная система UNIX. – http://www.citforum.ru/operating_systems/unix/contents.shtml.

[Олифер, 2000] Олифер В.Г., Олифер Н.А. Новые технологии и оборудование IP-сетей. – Спб.: BHV, 2000.

[Олифер, 2001] Олифер В.Г., Олифер Н.А. Сетевые операционные системы. – Спб.: Издательский дом Питер, 2001.

[Олифер, 2002] Олифер В.Г., Олифер Н.А. Компьютерные сети. Принципы, технологии, протоколы. – Спб.: Издательский дом Питер, 2002.

[Снейдер, 2001] Снейдер Й. Эффективное программирование TCP/IP. – Издательский дом Питер, 2001.

[Соломон, 2001] Соломон Д., Руссинович М. Внутреннее устройство Microsoft Windows 2000. – СПб: Издательский дом Питер, М.: Русская редакция, 2001.

[Стивенс, 2002] Стивенс У. UNIX: Взаимодействие процессов. – СПб: Издательский дом Питер, 2002.

[Стивенс, 2003] Стивенс У. UNIX: разработка сетевых приложений. – СПб: Издательский дом Питер, 2003.

[Столлингс, 2001] Столлингс В., Операционные системы. – М.: Вильямс, 2001.

[Таненбаум, 2002] Таненбаум Э. Современные операционные системы. – СПб.: Издательский дом Питер, 2002.

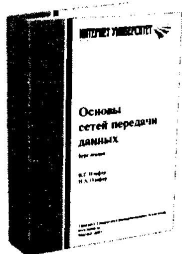
[Таненбаум, 2003] Таненбаум Э. Компьютерные сети. – СПб.: Издательский дом Питер, 2003.

[Таненбаум II, 2003] Таненбаум Э., Ван Стеен М. Распределенные системы. Принципы и парадигмы. – СПб.: Издательский дом Питер, 2003.

[Робачевский, 1999] Робачевский А. Операционная система UNIX. – Спб.: BHV, 1999.

[Цикритис, 1977] Цикритис Д., Бернстайн Ф. Операционные системы. – М.: Мир. 1977.

Серия «Основы информационных технологий»



Серия учебных пособий «Основы информационных технологий» открыта в издательстве Интернет-Университета Информационных Технологий в 2003 году и предполагает издание более 100 книг. В настоящее время в ее рамках вышли более 30 учебных пособий по самым разным направлениям информационных и коммуникационных технологий. Авторами этой серии являются известные профессора и преподаватели ведущих российских вузов, а также представители компьютерного бизнеса и академической среды. Ряд учебных курсов создаются при активном участии и поддержке ведущих отечественных и иностранных компаний, а также общественных организаций и коммерческих ассоциаций в области информационных технологий.

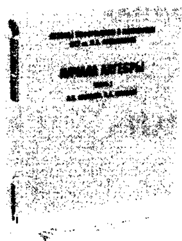
Книги серии

1. **Основы Web-технологий**,
   П.Б. Храмцов и др., 2003, 512 с., ISBN 5-9556-0001-9.
2. **Основы сетей передачи данных**,
   В.Г. Олифер, Н.А. Олифер, 2005, 176 с., ISBN 5-9556-0035-3.
3. **Основы информационной безопасности**, 2-е издание,
   В.А. Галатенко, 2004, 264 с., ISBN 5-9556-0015-9.
4. **Основы микропроцессорной техники**, 2-е издание,
   Ю.В. Новиков, П.К. Скоробогатов, 2004, 440 с.,
   ISBN 5-9556-0016-7.
5. **Язык программирования Си++**, 2-е издание,
   А.Л. Фридман, 2004, 264 с., ISBN 5-9556-0017-5.
6. **Программирование на Java**,
   Н.А. Вязовик, 2003, 592 с., ISBN 5-9556-0006-X.
7. **Стандарты информационной безопасности**,
   В.А. Галатенко, 2004, 328 с., ISBN 5-9556-0007-8.
8. **Основы функционального программирования**,
   Л.В. Городняя, 2004, 280 с., ISBN 5-9556-0008-6.
9. **Программирование в стандарте POSIX**,
   В.А. Галатенко, 2004, 560 с., ISBN 5-9556-0011-6.
10. **Введение в теорию программирования**,
    С.В. Зыков, 2004, 400 с., ISBN 5-9556-0009-4.
11. **Основы менеджмента программных проектов**,
    И.Н. Скопин, 2004, 336 с., ISBN 5-9556-0013-2.

12. **Основы операционных систем**, 2-е издание,
В.Е. Карпов, К.А. Коньков, 2004, 536 с., ISBN 5-9556-0044-2.
13. **Основы SQL**,
Л.Н. Полякова, 2004, 368 с., ISBN 5-9556-0014-0.
14. **Архитектуры и топологии многопроцессорных вычислительных систем**,
А.В. Богданов, В.В. Корхов, В.В. Мареев, Е.Н. Станкова, 2004, 176 с., ISBN 5-9556-0018-3.
15. **Операционная система UNIX**,
Г.В. Курячий, 2004, 320 с., ISBN 5-9556-0019-1.
16. **Основы сетевой безопасности: криптографические алгоритмы и протоколы взаимодействия**,
О.Р. Лапонина, 2005, 608 с., ISBN 5-9556-0020-5.
17. **Программирование в стандарте POSIX. Часть 2**,
В.А. Галатенко, 2005, 384 с., ISBN 5-9556-0021-3.
18. **Интеграция приложений на основе WebSphere MQ**,
В.А. Макушкин, Д.С. Володичев, 2005, 336 с., ISBN 5-9556-0031-0.
19. **Стили и методы программирования**,
Н.Н. Непейвода, 2005, 320 с., ISBN 5-9556-0023-X.
20. **Основы программирования на PHP**,
Н.В. Савельева, 2005, 264 с., ISBN 5-9556-0026-4.
21. **Основы баз данных**,
С.Д. Кузнецов, 2005, 488 с., ISBN 5-9556-0028-0.
22. **Интеллектуальные робототехнические системы**,
В.Л. Афонин, В.А. Макушкин, 2005, 208 с., ISBN 5-9556-0024-8.
23. **Программирование на языке Pascal**,
Т.А. Андреева, 2005, 240 с., ISBN 5-9556-0025-6.
24. **Основы тестирования программного обеспечения**,
В.П. Котляров, 2005, 360 с., ISBN 5-9556-0027-2.
25. **Язык Си и особенности работы с ним**
Н.И. Костюкова, Н.А. Калинина, 2005, 208с., ISBN 5-9556-0026-4.
26. **Основы локальных сетей**,
Ю.В. Новиков, С.В. Кондратенко, 2005, 360 с., ISBN 5-9556-0032-9.
27. **Операционная система Linux**,
Г.В. Курячий, К.А. Маслинский, 2005, 392 с., ISBN 5-9556-0029-9.
28. **Проектирование информационных систем**,
В.И. Грекул и др., 2005, 296 с., ISBN 5-9556-0033-7.
29. **Основы программирования на языке Пролог**,
П.А. Шрайнер, 2005, 176 с., ISBN 5-9556-0034-5.
30. **Операционная система Solaris**,
Ф.И. Торчинский, 2005, 472 с., ISBN 5-9556-0022-1.

продолжение следует

«Основы информатики и математики»



Новая серия учебных пособий по информатике и ее математическим основам открыта в 2005 году с целью современного изложения широкого спектра направлений информатики на базе соответствующих разделов математических курсов, а также примыкающих вопросов, связанных с информационными технологиями.

Особое внимание предполагается уделять возможности использования материалов публикуемых пособий в преподавании информатики и ее математических основ для непрофильных специальностей. Редакционная коллегия также надеется представить вниманию читателей широкую гамму практикумов по информатике и ее математическим основам, реализующих основные алгоритмы и идеи теоретической информатики.

Выпуск серии начат при поддержке корпорации Microsoft в рамках междисциплинарного научного проекта МГУ имени М.В. Ломоносова.

Книги серии

1. **Преподавание информатики и математических основ информатики**, под. ред. А.В. Михалева, 2005, 144 с., ISBN 5-9556-0037-Х.
2. **Начала алгебры, часть I**, А.В. Михалев, А.А. Михалев, 2005, 272 с., ISBN 5-9556-0038-8.
3. **Основы программирования**, В.В. Борисенко, 2005, 328 с., ISBN 5-9556-0039-6.
4. **Работа с текстовой информацией. Microsoft Office Word 2003**, О.Б. Калугина, В.С. Люцарев, 2005, 152 с., ISBN 5-9556-0041-8.

продолжение следует

Книги Интернет-Университета Информационных Технологий
всегда можно заказать на сайте: **www.intuit.ru**
Телефон: **(095) 253-9312**
e-mail: **admin@intuit.ru**
Адрес: Россия, Москва, 123056, Электрический пер., дом 8, строение 3

Серия «Основы информационных технологий»

В.Е. Карпов, К.А. Коньков
Под редакцией члена-корреспондента РАН
В.П. Иванникова

Основы операционных систем
Курс лекций. Учебное пособие
Издание второе, дополненное и исправленное

Редактор Е. Петровичева
Компьютерная верстка Ю. Волшмид
Корректор Ю. Голомазова
Обложка М. Автономова

Формат $60 \times 90^{1}/_{16}$. Усл. печ. л. 33,5. Бумага офсетная.
Подписано в печать 15.07.2005. Тираж 2000 экз. Заказ № 5523.

Санитарно-эпидемиологическое заключение о соответствии санитарным правилам №77.99.02.953.Д.006052.08.03 от 12.08.2003

ООО «ИНТУИТ.ру»
Интернет-Университет Информационных Технологий, www.intuit.ru
123056, Москва, Электрический пер., 8, стр. 3.

Отпечатано с готовых диапозитивов на ФГУП ордена «Знак Почета»
Смоленская областная типография им. В.И. Смирнова.
Адрес: 214000, г. Смоленск, пр-т им. Ю. Гагарина, д. 2.

© Интернет-Университет Информационных Технологий
www.intuit.ru, 2004-2005