Операционная система UNIX

Операционная система UNIX 1
Основание и история 6
Проект операционной системы Multics: неудача с положительными последствиями 6
Возникновение и первая редакция ОС UNIX 7
Исследовательский UNIX 8
Первый перенос ОС UNIX 9
Седьмая редакция 9
Возникновение группы университета г. Беркли (BSD) 10
UNIX System III и первые коммерческие версии системы 11
AT&T System V Release 2 и Release 3 11
Основные понятия ОС UNIX и базовые системные вызовы 14
Основные понятия 14
Ядро ОС UNIX 20
Файловая система 24
Разновидности файлов 29
Принципы защиты 35
Управление устройствами 37
Базовые механизмы сетевых взаимодействий 40
Распределенные файловые системы 46
Основные функции и компоненты ядра ОС UNIX 49
Управление памятью 49
Управление процессами и нитями 59
Управление вводом/выводом 71
Взаимодействие процессов 76
Мобильное программирование в среде ОС UNIX 90
Стандартные библиотеки 91
Мобильность на уровне исходных текстов 94
Бинарная совместимость 106
Традиционные средства интерактивного интерфейса пользователей 108
Командные языки и командные интерпретаторы 108
Команды и утилиты 114
Средства графического интерфейса пользователей 115
Оконная система X как базовое средство графических интерфейсов в среде ОС UNIX 117
Средства разработки графических интерфейсов 120
Современное состояние ОС UNIX 122
UNIX System V Release 4 и UnixWare 122
Системы, основанные на System V Release 4 123
Свободно распространяемые и коммерческие варианты ОС UNIX семейства BSD 125
Другие свободно распространяемые варианты ОС UNIX 125
Стандарты ОС UNIX 126
Перспективные ОС, поддерживающие среду ОС UNIX 128
Понятие микроядра 130
Микроядро Mach университета Карнеги-Меллон 131
Микроядро Chorus компании Chorus Systems 132
Примеры микроядерных реализаций ОС UNIX 133

Данное учебное пособие представляет собой информационно-аналитические материалы
Центра Информационных Технологий
Автор: С. Д. Кузнецов

http://athena.vvsu.ru/docs/unix/unix_citforum/contents.shtml.htm
http://www.citforum.ru/operating_systems/unix/contents.shtml

1. Основание и история
    - Проект операционной системы Multics: неудача с положительными последствиями
    - Возникновение и первая редакция ОС UNIX
    - Исследовательский UNIX
    - Первый перенос ОС UNIX
    - Седьмая редакция
    - Возникновение группы университета г. Беркли (BSD)
    - UNIX System III и первые коммерческие версии системы
    - AT&T System V Release 2 и Release 3
2. Основные понятия ОС UNIX и базовые системные вызовы
    - Основные понятия
        - Пользователь
        - Интерфейс пользователя
        - Привилегированный пользователь
        - Программы
        - Команды
        - Процессы
        - Перенаправление ввода/вывода
    - Ядро ОС UNIX
        - Общая организация традиционного ядра ОС UNIX
        - Основные функции
        - Принципы взаимодействия с ядром
        - Принципы обработки прерываний
    - Файловая система
        - Структура файловой системы
        - Монтируемые файловые системы
        - Интерфейс с файловой системой
    - Разновидности файлов
        - Обычные файлы
        - Файлы-каталоги
        - Специальные файлы
        - Связывание файлов с разными именами
        - Именованные программные каналы
        - Файлы, отображаемые в виртуальную память
        - Синхронизация при параллельном доступе к файлам
    - Принципы защиты
        - Идентификаторы пользователя и группы пользователей
        - Защита файлов
    - Управление устройствами
        - Устройство как специальный файл
        - Драйверы устройств
        - Внешний и внутренний интерфейсы устройств
    - Базовые механизмы сетевых взаимодействий
        - Потоки (Streams)

        - Стек протоколов TCP/IP
        - Программные гнезда (Sockets)
        - Вызовы удаленных процедур (RPC)
    - Распределенные файловые системы
        - Сетевая файловая система (NFS)
        - Совместное использование удаленных файлов (RFS)
3. Основные функции и компоненты ядра ОС UNIX
    - Управление памятью
        - Виртуальная память
        - Аппаратно-независимый уровень управления памятью
        - Страничное замещение основной памяти и swapping
    - Управление процессами и нитями
        - Пользовательская и ядерная составляющие процессов
        - Принципы организации многопользовательского режима
        - Традиционный механизм управления процессами на уровне пользователя
        - Понятие нити (threads)
        - Подходы к организации нитей и управлению ими в разных вариантах ОС UNIX
    - Управление вводом/выводом
        - Принципы системной буферизации ввода/вывода
        - Системные вызовы для управления вводом/выводом
        - Блочные драйверы
        - Символьные драйверы
        - Потоковые драйверы
    - Взаимодействие процессов
        - Разделяемая память
        - Семафоры
        - Очереди сообщений
        - Программные каналы
        - Программные гнезда (sockets)
        - Потоки (streams)
4. Мобильное программирование в среде ОС UNIX
    - Стандартные библиотеки
        - Библиотека системных вызовов
        - Библиотека ввода/вывода
        - Дополнительные библиотеки
        - Файлы заголовков
    - Мобильность на уровне исходных текстов
        - Особенности мобильного программирования на языке Си
        - Обеспечение независимости от особенностей версии ОС UNIX
    - Бинарная совместимость
        - Возможности достижения бинарной совместимости
        - Преимущества и ограничения
5. Традиционные средства интерактивного интерфейса пользователей
    - Командные языки и командные интерпретаторы
        - Общая характеристика командных языков
        - Базовые возможности семейства командных интерпретаторов
        - Bourne-shell
        - C-shell
        - Korn-shell
    - Команды и утилиты

        - Организация команды в ОС UNIX
        - Перенаправление ввода/вывода и организация конвейера
        - Встроенные, библиотечные и пользовательские команды
        - Программирование на командном языке
6. Средства графического интерфейса пользователей
    - Оконная система X как базовое средство графических интерфейсов в среде ОС UNIX
        - Общая организация X-Window
        - Клиентская и серверная части
        - Базовые библиотеки
    - Средства разработки графических интерфейсов
        - Пакет Motif
        - Язык и интерпретатор Tcl/Tk
7. Современное состояние ОС UNIX
    - UNIX System V Release 4 и UnixWare
    - Системы, основанные на System V Release 4
        - Solaris компании Sun Microsystems
        - HP/UX компании Hewlett-Packard, DG/UX компании Data General, AIX компании IBM
        - Santa Cruz Operation и SCO UNIX
        - Open Software Foundation и OSF-1
    - Свободно распространяемые и коммерческие варианты ОС UNIX семейства BSD
    - Другие свободно распространяемые варианты ОС UNIX
        - Linux университета Хельсинки
        - Hurd Free Software Foundation
    - Стандарты ОС UNIX
        - System V Interface Definition (SVID)
        - Деятельность комитетов POSIX
        - Деятельность X/Open
        - Стандарт ANSI C
        - Другие стандарты
8. Перспективные ОС, поддерживающие среду ОС UNIX
    - Понятие микроядра
    - Микроядро Mach университета Карнеги-Меллон
    - Микроядро Chorus компании Chorus Systems
    - Примеры микроядерных реализаций ОС UNIX
        - OSF-1 компании Open Software Foundation
        - MiX компании Chorus Systems
        - Hurd Free Software Foundation

Основание и история

История ОС UNIX началась в недрах Bell Telephone Laboratories (теперь AT&T Bell Laboratories) и связана с известными теперь всем именами Кена Томпсона, Денниса Ритчи и Брайана Кернигана (два последних имени знакомы читателям и как имена авторов популярнейшей книги по языку программирования Си, издававшейся в нашей стране на русском языке).

Проект операционной системы Multics: неудача с положительными последствиями

С 1965 по 1969 год компания Bell Labs совместно с компанией General Electric и группой исследователей из Масачусетского технологического института участвовала в проекте ОС Multics. Целью проекта было создание многопользовательской интерактивной операционной системы, обеспечивающей большое число пользователей удобными и мощными средствами доступа к вычислительным ресурсам. В этом курсе мы не ставим задачу познакомить слушателей с ОС Multics. Это могло бы быть темой отдельного большого курса. Однако отметим хотя бы некоторые идеи, которые содержались в проекте MAC (так назывался проект ОС Multics).

Во-первых, эта система основывалась на принципах многоуровневой защиты. Виртуальная память имела сегментно-страничную организацию, разделялись сегменты данных и сегменты программного кода, и с каждым сегментом связывался уровень доступа (по выполнению для сегментов команд и уровень чтения и записи для сегментов данных). Для того, чтобы какая-либо программа могла вызвать программу или обратиться к данным, располагающимся в некотором сегменте, требовалось, чтобы уровень выполнения этой программы (точнее, сегмента, в котором эта программа содержалась, был не ниже уровня доступа соответствующего сегмента). Такая организация позволяла практически полностью и с полной защитой содержать операционную систему в системных сегментах любого пользовательского виртуального адресного пространства.

Во-вторых, в ОС Multics была спроектирована и реализована полностью централизованная файловая система. В централизованной файловой системе файлы, физически располагающиеся на разных физических устройствах внешней памяти, логически объединяются в один централизованный архив или древовидную иерархическую структуру, промежуточными узлами которой являются именованные каталоги, а в листьях содержатся ссылки на файлы. В том случае, когда при поиске файла в архиве по его имени оказывалось, что соответствующий накопитель (магнитный диск или магнитная лента) не был установлен на устройство внешней памяти, ОС обращалась к оператору с требованием установить нужный том внешней памяти. Естественно, такая дисциплина существенно облегчала операторскую работу и администрирование файловой системы, хотя и затрудняла выполнение таких рутинных действий как перенос части файловой системы с одного компьютера на другой. Позже мы увидим, какой изящный компромисс был выбран при реализации ОС UNIX.

Далее, наличие большой сегментно-страничной виртуальной памяти позволило использовать отображение файлов в сегменты виртуальной памяти. Другими словами, при открытии файла в виртуальной памяти соответствующего процесса образовывался сегмент, в который полностью отображался файл, располагающийся во внешней памяти. (Следует отметить, что в файловой системе ОС Multics на базовом уровне поддерживались файлы со страничной структурой. Более сложные организации являлись

надстройкой). Дальнейшая работа с файлом происходила на основе общего механизма управления виртуальной памятью.

Операционная система Multics, хотя и не была полностью доведена до стадии коммерческого продукта, обогатила мировое сообщество системных программистов массой ценных идей, многие из которых сохраняют свою актуальность по сей день и используются применительно не только к операционным системам. Основным недостатком ОС Multics, который, по всей видимости, и помешал довести систему до уровня программного продукта, была ее чрезмерная сложность. Среди участников проекта Multics находились Кен Томпсон и Деннис Ритчи.

Решение о прекращении участия в проекте Multics было принято на самом верхнем уровне руководства Bell Labs, и сотрудники, по существу, были поставлены перед свершившимся фактом. Более того, руководство компании, разочарованное результатами весьма дорогостоящего проекта, вообще не желало больше вести какие-либо работы, связанные с операционными системами.

Возникновение и первая редакция ОС UNIX

Принято считать, что исходным толчком к появлению ОС UNIX явилась работа Кена Томпсона по созданию компьютерной игры "Space Travel". Он делал это в 1969 году на компьютере Honeywell 635, который до этого использовался для разработки проекта MAC. В это же время Кен Томпсон, Деннис Ритчи и другие сотрудники Bell Labs предложили идею усовершенствованной файловой системы, прототип которой был реализован на компьютере General Electric 645. Однако компьютер GE-645, который был рассчитан на работу в режиме разделения времени и не обладал достаточной эффективностью, не годился для переноса Space Travel. Томпсон стал искать замену и обнаружил, что появившийся к этому времени 18-разрядный компьютер PDP-7 с 4 килословами оперативной памяти и качественным графическим дисплеем вполне для этого подходит.

После того, как игра была успешно перенесена на PDP-7, Томпсон решил реализовать на PDP-7 разработанную ранее файловую систему. Дополнительным основанием для этого решения было то, что компания Bell Labs испытывала потребность в удобных и дешевых средствах подготовки и ведения документации. В скором времени на PDP-7 работала файловая система, в которой поддерживались: понятие inodes, подсистема управления процессами и памятью, обеспечивающая использование системы двумя пользователями в режиме разделения времени, простой командный интерпретатор и несколько утилит. Все это еще не называлось операционной системой UNIX, но уже содержало родовые черты этой ОС.

Название придумал Брайан Керниган. Он предложил назвать эту двухпользовательскую систему UNICS (Uniplexed Information and Computing System). Название понравилось, поскольку, помимо прочего, оно напоминало об участии сотрудников Bell Labs в проекте Multics. В скором времени UNICS превратилось в UNIX (произносится так же, но на одну букву короче).

Первыми реальными пользователями UNIX стали сотрудники патентного отдела Bell Labs. Однако существовали некоторые проблемы, связанные с PDP-7. Во-первых, эта машина не принадлежала компьютерной группе (была только во временном пользовании). Во-вторых, возможности этого компьютера не удовлетворяли потребности исследователей. Поэтому в 1971 году был приобретен новый 16-разрядный компьютер

фирмы Digital Equipment PDP-11/20, и на него была перенесена UNIX. Существовавший к этому времени вариант системы был написан на языке ассемблера, так что можно представить, что перенос был совсем не простым делом. На PDP-11 система поддерживала большее число пользователей. Кроме того, была реализована утилита форматирования текстовых документов roff (тоже на языке ассемблера).

В ноябре 1971 года был опубликован первый выпуск документации по ОС UNIX ("Первая редакция"). В соответствии с этой "Первой редакцией" назвали и соответствующий документации вариант системы. Впоследствии это стало традицией: новая редакция ОС UNIX объявлялась при выходе в свет новой редакции документации.

Вторая редакция появилась в 1972 году. Наиболее существенным качеством "Второй редакции" было то, что система была переписана на языке Би ("B"). Язык и интерпретирующая система программирования были разработаны Кеном Томпсоном под влиянием существовавшего языка BCPL. Во второй редакции появились программные каналы ("pipes").

Появление варианта системы, написанного не на языке ассемблера, было заметным продвижением. Однако сам язык Би во многом не удовлетворял разработчиков. Подобно языку BCPL язык Би был бестиповым, в нем поддерживался только один тип данных, соответствующий машинному слову. Другие типы данных эмулировались библиотекой функций. Деннис Ритчи, который всегда увлекался языками программирования, решил устранить ограничения языка Би, добавив в язык систему типов. Так возник язык Си ("C"). В 1973 году Томпсон и Ритчи переписали систему на языке Си. К этому времени существовало около 25 установок ОС UNIX, и это была "Четвертая редакция".

В июле 1974 года Томпсон и Ритчи опубликовали в журнале Communications of the ACM историческую статью "UNIX Timesharing Operating System", которая положила начало новому этапу в истории системы. ОС UNIX заинтересовались в университетах. Этому способствовала политика компании Bell Labs, которая объявила о возможности бесплатного получения исходных текстов UNIX для использования в целях образования (нужно было платить только за носитель и документацию).

Появившуюся к этому времени "Пятую редакцию" ОС UNIX одними из первых получили Калифорнийский университет г. Беркли и университет Нового Южного Уэльса г. Сидней (Австралия).

Исследовательский UNIX

В 1975 году компания Bell Labs выпустила "Шестую редакцию" ОС UNIX, известную как V6 или Исследовательский UNIX. Эта версия системы была первой коммерчески доступной вне Bell Labs. К этому времени большая часть системы была написана на языке Си. Небольшие размеры языка и наличие сравнительно легко переносимого компилятора придавали ОС UNIX V6 новое качество реально переносимой операционной системы. Кроме того, потенциальное наличие на разных аппаратных платформах компилятора языка Си делало возможным разработку мобильного прикладного программного обеспечения.

Важный шаг в этом направлении был предпринят Деннисом Ритчи, который в 1976 году создал библиотеку ввода/вывода (stdio), ставшую фактическим стандартом различных систем программирования на языке Си. С использованием stdio стало возможно создавать

мобильные прикладные программы, действительно независящие от особенностей аппаратуры процессора и внешних устройств.

Примерно в это же время Кен Томпсон во время своего академического отпуска посетил университет г. Беркли и установил там UNIX V6 на компьютере PDP-11/70. Билл Джой (основатель BSD - Berkeley Software Distribution, а впоследствии основатель и вице-президент компании Sun Microsystems) был тогда дипломником этого университета.

Первый перенос ОС UNIX

По-видимому, первый перенос ОС UNIX на компьютер с архитектурой, принципиально отличающейся от PDP-11, был произведен в 1977 году в Австралии. Это произошло вскоре после того, как в университете Воллонгонга была образована компьютерная кафедра. Джюрис Рейндфельдс, ставший заведующим новой кафедры, решил использовать ОС UNIX как основу обучения студентов. Он специально посетил университет г. Беркли и был вдохновлен возможностями, имеющимися в этом университете (PDP-11/40 с ОС UNIX V6). Однако выяснилось, что в университете г.Воллонгонг отсутствовали средства, достаточные для приобретения PDP-11.

Профессор Рейндфельдс был вынужден купить 32-разрядный компьютер Interdata 7/32, который был существенно дешевле, хотя и слабее по производительности. После нескольких попыток здравым образом дополнить "родную" операционную систему Interdata 7/32 OSMT/32 более развитыми средствами многопользовательского режима использования было принято решение попробовать перенести на эту 32-разрядную машину ОС UNIX V6.

Очень замысловатым образом (напомним, что в австралийском университете не было доступного компьютера PDP-11) путем обмена магнитными лентами с университетом г. Беркли Ричард Миллер (канадец, работавший в Австралии) смог к январю 1977 года получить компилятор языка Си, который мог успешно компилировать собственный исходный текст на Interdata 7/32. Это позволило уже через месяц получить некоторый вариант ОС UNIX, работающий на этой же машине.

Система Миллера представляла собой некий гибрид, основанный на ОС UNIX V6 и выполняемый "поверх" OSMT/32. Версия системы не включала собственных средств управления терминалами и обработки прерываний и поддерживала около восьми команд примитивного командного интерпретатора. Тем не менее, это была первая успешная (и быстро выполненная) попытка переноса ОС UNIX на компьютер с 32-разрядной архитектурой.

Седьмая редакция

После завершения своей работы Ричард Миллер отправился в Bell Labs с целью обсудить полученные результаты с Томпсоном и Ритчи. Незадолго до этого в Bell Labs был закуплен компьютер Interdata 8/32 (модель, следующая за Interdata 7/32). В принципе, компания Bell Labs была удовлетворена возможностями и ценой компьютеров семейства PDP-11. Однако 16-разрядная организация этих компьютеров ограничивала возможности ОС UNIX (слишком малый размер виртуальной памяти для разработки больших и сложных программ). Переход на 32-разрядные архитектуры позволял преодолеть эти ограничения.

Наличие 32-разрядного компьютера Interdata 8/32 и имеющийся положительный опыт Ричарда Миллера по переносу (хотя и не полному) ОС UNIX на Interdata привели к тому, что Томпсон и Ритчи решили произвести полный перенос UNIX на свою новую машину. Для начала требовалось развить язык Си, чтобы программисты могли использовать особенности 32-разрядных архитектур. Для этого Деннис Ритчи расширил систему типов языка Си типами union, short integer, long integer и unsigned integer. В дополнение к этому, в языке появились развитые средства инициализации переменных, битовые поля, макросы и средства условной компиляции, регистровые и глобальные переменные и т.д. Одним словом, язык Си стал таким, каким он описан в известнейшей книге Кернигана и Ритчи "Язык программирования Си" (сокращенно принято называть этот диалект языка K&R).

Однако одного расширенного языка Си было недостаточно для переноса UNIX, поскольку сама организация UNIX V6 была слишком ориентирована на особенности PDP-11. Пришлось полностью переписать подсистему управления оперативной и виртуальной памятью и изменить интерфейс драйверов внешних устройств, чтобы сделать систему более легко переносимой на другие архитектуры. Результатом работы стала "Седьмая редакция" UNIX (чаще ее называют UNIX Version 7). В состав новой версии системы входил компилятор нового диалекта языка Си PCC (Portable C-Compiler), новый командный интерпретатор sh, называемый также в честь своего создателя Bourne-shell, набор новых драйверов устройств и многое другое.

После выпуска UNIX Version 7 Деннис Ритчи поехал на конференцию в Австралию и взял с собой магнитную ленту с исходными текстами системы. В Мельбурнском университете был осуществлен полный перенос системы на Interdata 8/32. Позднее в Воллонгонге система была повторно перенесена на Interdata 7/32. Таким образом, в результате совместной плодотворной работы исследователей из США и Австралии было продемонстрировано одно из наиболее ярких качеств ОС UNIX - мобильность. Кроме того, стало ясно, что полезно привлекать к работе над ОС UNIX сотрудников и студентов университетов.

Возникновение группы университета г. Беркли (BSD)

Как мы упоминали выше, в 1976 году Кен Томпсон провел свой академический отпуск в университете г. Беркли и принял участие в проводившихся там исследованиях. Это привело к возникновению серьезного интереса к ОС UNIX среди профессоров и студентов. Появились местные знатоки системы, среди которых одним из наиболее сильных был Билл Джой.

Билл Джой собрал вместе с целью дальнейшего распространения большой объем программного обеспечения, включавший полный набор текстов UNIX V6, компилятор языка Паскаль, свой собственный редактор `ex` (потом его стали называть `vi`) и другие программы. Все это было названо Berkeley Software Distribution (BSD 1.0). Вокруг BSD сложилась небольшая, но очень сильная группа молодых программистов. Бытует мнение, что именно группа BSD смогла добиться практически полного устранения ошибок в UNIX V6. Не будучи удовлетворенной структурой и функциями ядра UNIX V6, группа BSD в своем втором выпуске (BSD 2.x) предприняла серьезную попытку переписать ядро системы.

В компьютерном отделении университета Беркли имелось несколько компьютеров семейства VAX компании Digital. Группа BSD при участии сотрудников Bell Labs Джона Рейзера и Тома Лондона произвела перенос UNIX Version 7 на 32-разрядную архитектуру VAX. Этот вариант UNIX назывался 32/V. В ядре системы появились новые свойства

страничного замещения оперативной памяти и управления виртуальной памятью. Система стала основой третьего выпуска - BSD 3.x.

В группе BSD был разработан и впервые реализован стек транспортных протоколов TCP/IP (Transport Control Protocol/Internet Protocol). Эта работа финансировалась министерством безопасности США.

Bell Labs и университет Беркли заключили соглашение, в соответствии с которым группа BSD могла распространять свои версии ОС UNIX среди любых пользователей, которые располагали лицензией Bell Labs. Если учесть, что UNIX BSD исторически распространялся бесплатно (с исходными текстами!), а лицензия Bell Labs к этому времени стоила уже весьма недешево, то можно понять группу BSD, которая, начиная с первой версии BSD 4.1 (1980 год), стремилась к тому, чтобы освободить пользователей UNIX BSD от необходимости приобретать лицензию Bell Labs. Подробности этого процесса и возникшие коллизии мы рассмотрим в разделе, посвященном современному состоянию ОС UNIX.

UNIX System III и первые коммерческие версии системы

В 1978 году в Bell Labs специально для поддержки ОС UNIX была организована Группа поддержки ОС UNIX (UNIX Support Group - USG). Эта группа выпустила несколько версий системы, но они не имели хождения за пределами Bell Labs.

Однако, к этому времени большой интерес к ОС UNIX стали проявлять коммерческие компании-производители компьютеров и программного обеспечения. Это объясняется тем, что с развитием технологии электронных схем резко упала стоимость производства новых однокристальных процессоров. Поэтому наличие по-настоящему мобильной операционной системы, перенос которой на новую аппаратную платформу не занимал слишком много времени и средств, позволяло экономно оснастить новые компьютеры качественным базовым программным обеспечением. Появились компании, специализирующиеся на переносе UNIX на новые платформы.

Одной из первых была компания UniSoft Corporation, которая производила свою версию UNIX под названием UniPlus+. Microsoft Corporation совместно с Santa Cruz Operation (SCO) произвели вариант UNIX под названием XENIX. В результате к концу 70-х UNIX-подобные операционные системы были доступны на компьютерах, основанных на микропроцессорах Zilog, Intel, Motorola и т.д. Появились тысячи установок с ОС UNIX.

В 1982 году USG выпустила за пределы Bell Labs свой первый вариант UNIX, получивший название UNIX System III. В этой системе сочетались лучшие качества UNIX Version 7, V/32 и других вариантов UNIX, имевших хождение в Bell Labs.

AT&T System V Release 2 и Release 3

В начале 1983 года компания American Telephone and Telegraph Bell Laboratories (AT&T Bell Labs) объявила о выпуске UNIX System V. Впервые в истории Bell Labs было также объявлено, что AT&T будет поддерживать этот и все будущие выпуски System V. Кроме того, была обещана совместимость выпущенной версии System V со всеми будущими версиями. ОС UNIX System V включала много новых возможностей, но почти все они относились к повышению производительности (хеш-таблицы и кэширование данных). На самом деле UNIX System V являлась развитым вариантом UNIX System III. К наиболее

важным оригинальным особенностям UNIX System V относится появление семафоров, очередей сообщений и разделяемой памяти.

В 1984 году USG была преобразована в Лабораторию по развитию системы UNIX (UNIX System Development Laboratories - USDL). В 1984 году USDL выпустила UNIX System V Release 2 (SVR2). В этом варианте системы появились возможности блокировок файлов и записей, копирования совместно используемых страниц оперативной памяти при попытке записи (copy-on-write), страничного замещения оперативной памяти (реализованного не так, как в BSD) и т.д. К этому времени ОС UNIX была установлена на более чем 100000 компьютеров.

В 1987 году подразделение USDL объявило о выпуске UNIX System V Release 3 (SVR3). В этой системе появились полные возможности межпроцессных взаимодействий, разделения удаленных файлов (Remote File Sharing - RFS), развитые операции обработки сигналов, разделяемые библиотеки и т.д. Кроме того, были обеспечены новые возможности по повышению производительности и безопасности системы. К концу 1987 года появилось более 750000 установок ОС UNIX, и было зарегистрировано 4,5 млн. пользователей.

На этом мы заканчиваем исторический обзор ОС UNIX, поскольку вплотную подошли к современному состоянию системы. Продолжим этот разговор в конце курса, а пока ограничимся таблицей 1.1 и рисунком генеалогического дерева ОС UNIX (заметим, что по поводу генеалогии существуют разные мнения).

Таблица 1.1.
Характерные свойства версий AT&T UNIX начиная с 1982 года

1982 System III	Именованные программные каналы
	Очереди запуска
1983 System V	Хеш-таблицы
	Кэши буферов и inodes
	Семафоры
	Разделяемая память
	Очереди сообщений
1984 SVR2	Блокирование записей и файлов
	Подкачка по требованию
	Копирование по записи
1987 SVR3	Межпроцессные взаимодействия (IPC)
	Разделение удаленных файлов (RFS)
	Развитые операции обработки сигналов
	Разделяемые библиотеки
	Переключатель файловых систем (FSS)
	Интерфейс транспортного уровня (TLI)
	Возможности коммуникаций на основе потоков
1989 SVR4	Поддержка обработки в реальном времени
	Классы планирования процессов

	Динамически выделяемые структуры данных
	Развитые возможности открытия файлов
	Управление виртуальной памятью (VM)
	Возможности виртуальной файловой системы (VFS)
	Быстрая файловая система (BSD)
	Развитые возможности потоков
	Прерываемое ядро
	Квоты файловых систем
	Интерфейс драйвера с ядром системы

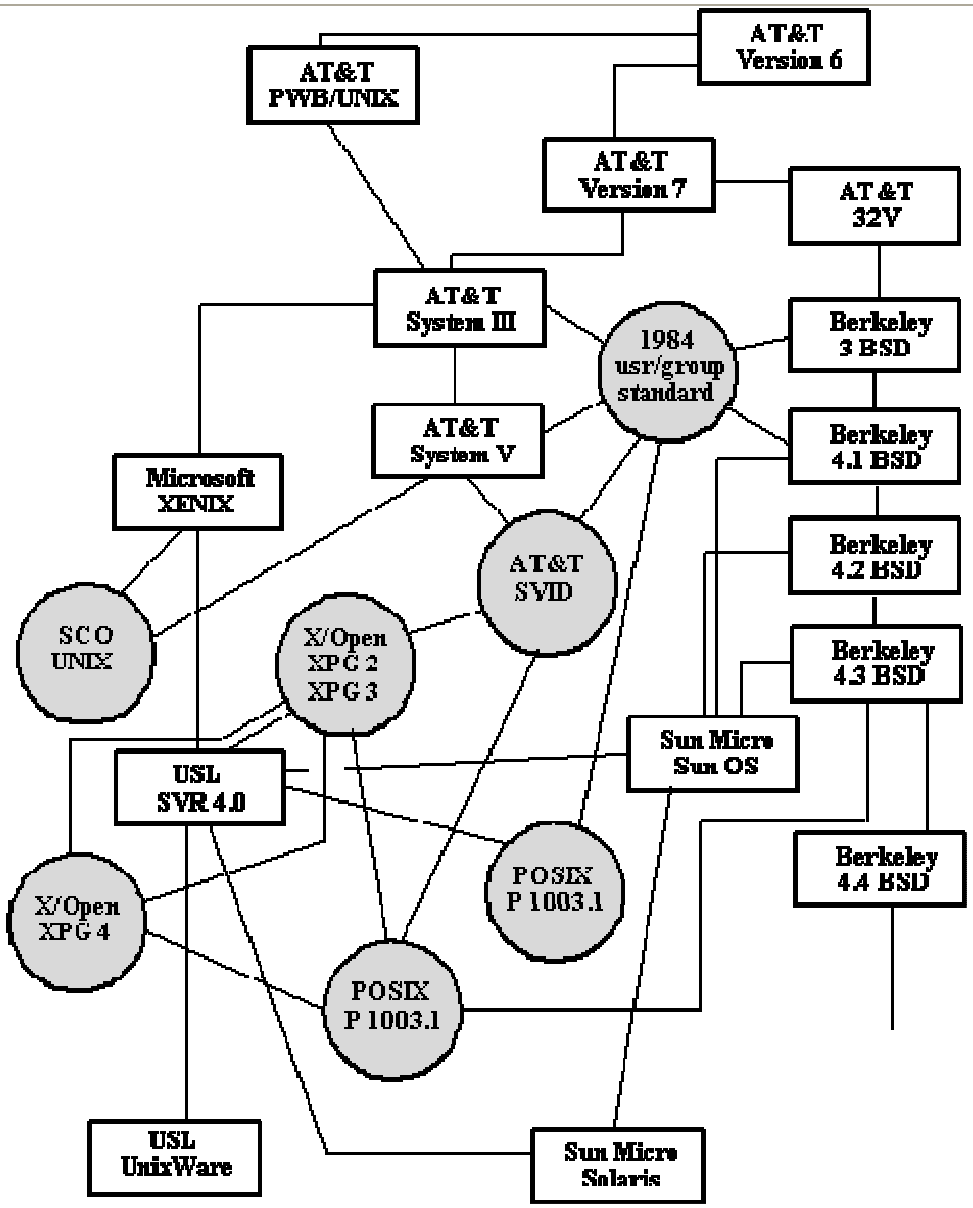

Рис. 1.1. Генеалогическое дерево ОС UNIX

Основные понятия ОС UNIX и базовые системные вызовы

В этой части курса вводятся основные понятия, на которые опирается ОС UNIX, рассматривается общая структура системы и обсуждаются ее основные возможности.

Основные понятия

Одним из достоинств ОС UNIX является то, что система базируется на небольшом числе интуитивно ясных понятий. Однако, несмотря на простоту этих понятий, к ним нужно привыкнуть. Без этого невозможно понять существо ОС UNIX.

Пользователь

С самого начала ОС UNIX замышлялась как интерактивная система. Другими словами, UNIX предназначен для терминальной работы. Чтобы начать работать, человек должен "войти" в систему, введя со свободного терминала свое учетное имя (account name) и, возможно, пароль (password). Человек, зарегистрированный в учетных файлах системы, и, следовательно, имеющий учетное имя, называется зарегистрированным пользователем системы. Регистрацию новых пользователей обычно выполняет администратор системы. Пользователь не может изменить свое учетное имя, но может установить и/или изменить свой пароль. Пароли хранятся в отдельном файле в закодированном виде. Не забывайте свой пароль, снова узнать его не поможет даже администратор!

Все пользователи ОС UNIX явно или неявно работают с файлами. Файловая система ОС UNIX имеет древовидную структуру. Промежуточными узлами дерева являются каталоги со ссылками на другие каталоги или файлы, а листья дерева соответствуют файлам или пустым каталогам. Каждому зарегистрированному пользователю соответствует некоторый каталог файловой системы, который называется "домашним" (home) каталогом пользователя. При входе в систему пользователь получает неограниченный доступ к своему домашнему каталогу и всем каталогам и файлам, содержащимся в нем. Пользователь может создавать, удалять и модифицировать каталоги и файлы, содержащиеся в домашнем каталоге. Потенциально возможен доступ и ко всем другим файлам, однако он может быть ограничен, если пользователь не имеет достаточных привилегий.

Интерфейс пользователя

Традиционный способ взаимодействия пользователя с системой UNIX основывается на использовании командных языков (правда, в настоящее время все большее распространение получают графические интерфейсы). После входа пользователя в систему для него запускается один из командных интерпретаторов (в зависимости от параметров, сохраняемых в файле `/etc/passwd`). Обычно в системе поддерживается несколько командных интерпретаторов с похожими, но различающимися своими возможностями командными языками. Общее название для любого командного интерпретатора ОС UNIX - `shell` (оболочка), поскольку любой интерпретатор представляет внешнее окружение ядра системы.

Вызванный командный интерпретатор выдает приглашение на ввод пользователем командной строки, которая может содержать простую команду, конвейер команд или последовательность команд. После выполнения очередной командной строки и выдачи на экран терминала или в файл соответствующих результатов, `shell` снова выдает

приглашение на ввод командной строки, и так до тех пор, пока пользователь не завершит свой сеанс работы путем ввода команды logout или нажатием комбинации клавиш Ctrl-d.

Командные языки, используемые в ОС UNIX, достаточно просты, чтобы новые пользователи могли быстро начать работать, и достаточно мощны, чтобы можно было использовать их для написания сложных программ. Последняя возможность опирается на механизм командных файлов (`shell scripts`), которые могут содержать произвольные последовательности командных строк. При указании имени командного файла вместо очередной команды интерпретатор читает файл строка за строкой и последовательно интерпретирует команды.

Привилегированный пользователь

Ядро ОС UNIX идентифицирует каждого пользователя по его идентификатору (UID - User Identifier), уникальному целому значению, присваиваемому пользователю при регистрации в системе. Кроме того, каждый пользователь относится к некоторой группе пользователей, которая также идентифицируется некоторым целым значением (GID - Group IDentifier). Значения UID и GID для каждого зарегистрированного пользователя сохраняются в учетных файлах системы и приписываются процессу, в котором выполняется командный интерпретатор, запущенный при входе пользователя в систему. Эти значения наследуются каждым новым процессом, запущенным от имени данного пользователя, и используются ядром системы для контроля правомощности доступа к файлам, выполнения программ и т.д.

Понятно, что администратор системы, который, естественно, тоже является зарегистрированным пользователем, должен обладать большими возможностями, чем обычные пользователи. В ОС UNIX эта задача решается путем выделения одного значения UID (нулевого). Пользователь с таким UID называется суперпользователем (`superuser`) или `root`. Он имеет неограниченные права на доступ к любому файлу и на выполнение любой программы. Кроме того, такой пользователь имеет возможность полного контроля над системой. Он может остановить ее и даже разрушить.

В мире UNIX считается, что человек, получивший статус суперпользователя, должен понимать, что делает. Суперпользователь должен хорошо знать базовые процедуры администрирования ОС UNIX. Он отвечает за безопасность системы, ее правильное конфигурирование, добавление и исключение пользователей, регулярное копирование файлов и т.д.

Еще одним отличием суперпользователя от обычного пользователя ОС UNIX является то, что на суперпользователя не распространяются ограничения на используемые ресурсы. Для обычных пользователей устанавливаются такие ограничения как максимальный размер файла, максимальное число сегментов разделяемой памяти, максимально допустимое пространство на диске и т.д. Суперпользователь может изменять эти ограничения для других пользователей, но на него они не действуют.

Программы

ОС UNIX одновременно является операционной средой использования существующих прикладных программ и средой разработки новых приложений. Новые программы могут писаться на разных языках (Фортран, Паскаль, Модула, Ада и др.). Однако стандартным языком программирования в среде ОС UNIX является язык Си (который в последнее время все больше заменяется на Си++). Это объясняется тем, что во-первых, сама система

UNIX написана на языке Си, а, во-вторых, язык Си является одним из наиболее качественно стандартизованных языков.

Поэтому программы, написанные на языке Си, при использовании правильного стиля программирования обладают весьма высоким уровнем мобильности, т.е. их можно достаточно просто переносить на другие аппаратные платформы, работающие как под управлением ОС UNIX, так и под управлением ряда других операционных систем (например, DEC Open VMS или MS Windows NT). Более подробно мы рассмотрим принципы мобильного программирования в среде ОС UNIX в четвертой части курса.

Приведем краткий обзор процесса разработки программы на языке Си (или Си++), которую можно выполнить в среде ОС UNIX. Любая выполняемая программа компонуется из одного или нескольких объектных файлов. Поэтому разработка программы начинается с создания исходных файлов, содержащих текст на языке Си. Эти файлы могут содержать определения глобальных имен переменных и/или функций (имен, которые могут быть видимы из других файлов), а также ссылки на внешние имена (объявленные как глобальные в одном из других файлов, которые будут составлять программу).

Текстовые файлы производятся с помощью одного из текстовых редакторов, поддерживаемых в среде UNIX. Традиционным текстовым редактором ОС UNIX является упоминавшийся в первом разделе редактор `vi`, исходная версия которого была разработана Биллом Джоем. Этот редактор достаточно старый, он может работать практически на всех терминалах и не является в полном смысле оконным.

В последние годы все большую популярность получает редактор `Emacs` (разработанный и непрерывно совершенствуемый президентом Free Software Foundation Ричардом Столлманом). Это очень мощный многооконный редактор, который позволяет не только писать программы (и другие тексты), но также и компилировать, компоновать и отлаживать программы (а также делать многое другое, например, принимать и отправлять электронную почту). Основным недостатком редактора `Emacs` является исключительно большой набор (более 200) функциональных клавиш. Следует, правда, заметить, что при использовании Emacs в оконной системе X он обеспечивает более удобный интерфейс.

Заметим также, что многие неудобства интерфейсов традиционных инструментальных средств ОС UNIX связаны с тем, что они ориентированы на использование и алфавитно-цифровых, и графических терминалов. Поэтому обычно эти средства поддерживают старомодный строчный интерфейс даже при наличии графического терминала. Естественно, в современных вариантах ОС UNIX все новые инструментальные средства поддерживают оконный графический интерфейс (и, следовательно, их невозможно использовать при наличии алфавитно-цифровых терминалов).

После того, как текстовый файл создан, его нужно откомпилировать для получения объектного файла. Наиболее популярными компиляторами для языка Си в среде ОС UNIX сейчас являются `pcc` (Ритчи и Томпсон) и `gcc` (Ричард Столлман). Оба эти компилятора являются полностью мобильными и обладают возможностью генерировать код для разнообразных компьютеров, т.е. эти компиляторы могут быть установлены практически на любой аппаратной платформе под управлением ОС UNIX.

Можно отметить следующие преимущества `gcc`. Во-первых, этот компилятор свободно, т.е. бесплатно (вместе со своими исходными текстами) распространяется Free Software Foundation. Во-вторых, `gcc` тщательно поддерживается и сопровождается. В-третьих,

начиная с версии 2.0, `gcc` может компилировать программы, написанные на языках Си, Си++ и Objective C, а результирующая выполняемая программа может быть скомпонована из объектных файлов, полученных из текстовых файлов на любом из этих языков. В-четвертых, открытость исходных текстов `gcc` и тщательно разработанная структура компилятора позволяют сравнительно просто добавлять к `gcc` новые кодогенераторы. Относительным недостатком `gcc` является то, что используемый диалект языка Си включает слишком много расширений по сравнению со стандартом ANSI/ISO (однако имеется режим, в котором компилятор указывает все расширенные конструкции языка, встречающиеся в компилируемой программе).

Оба компилятора обрабатывают программу в два этапа. На первом этапе синтаксически правильный текст на языке Си преобразуется в текст на языке ассемблера. На втором этапе на основе текста на языке ассемблера генерируются машинные коды и получается объектный файл. Исторически в ОС UNIX использовались различные форматы объектных модулей. Для обеспечения совместимости с предыдущими версиями почти все они поддерживаются в современных версиях компиляторов. Однако в настоящее время преимущественно используется формат COFF (Common Object File Format). При желании можно остановить процесс компиляции после первого этапа и получить для изучения файл с текстом программы на языке ассемблера.

После того, как необходимый для построения выполняемой программы набор объектных файлов получен, необходимо произвести компоновку выполняемой программы. В ОС UNIX компоновщик выполняемых программ называется редактором связей (`link editor`) и обычно вызывается командой `ld`. Редактору связей указывается набор объектных файлов и набор библиотек, из которых нужно черпать недостающие для компоновки программы.

Процесс компоновки заключается в следующем. Сначала просматривается набор заданных объектных файлов. Для каждого внешнего имени ищется объектный файл, содержащий определение такого же глобального имени. Если поиск заканчивается успешно, то внешняя ссылка заменяется на ссылку на определение глобального имени. Если в конце этого этапа остаются внешние имена, для которых не удалось найти соответствующего определения глобального имени, то начинается поиск объектных файлов с нужными определениями глобальных имен в указанных библиотеках. Если, в конце концов, удается найти определения для всех внешних имен, все соответствующие объектные файлы собираются вместе и образуют выполняемый файл.

В ОС UNIX имеется несколько стандартных библиотек. В большинстве случаев наиболее важной является библиотека ввода/вывода (`stdio`). Грамотное использование стандартных библиотек способствует созданию легко переносимых прикладных программ (мы вернемся к обсуждению стандартных библиотек ОС UNIX в четвертой части курса).

Выполняемая программа может быть запущена в интерактивном режиме как команда `shell` или выполнена в отдельном процессе, образуемом уже запущенной программой.

Команды

Любой командный язык семейства `shell` фактически состоит из трех частей: служебных конструкций, позволяющих манипулировать с текстовыми строками и строить сложные команды на основе простых команд; встроенных команд, выполняемых непосредственно интерпретатором командного языка; команд, представляемых отдельными выполняемыми

файлами (более подробно и точно командные языки рассматриваются в пятой части курса).

В свою очередь, набор команд последнего вида включает стандартные команды (системные утилиты, такие как `vi`, `cc` и т.д.) и команды, созданные пользователями системы. Для того, чтобы выполняемый файл, разработанный пользователем ОС UNIX, можно было запускать как команду `shell`, достаточно определить в одном из исходных файлов функцию с именем `main` (имя `main` должно быть глобальным, т.е. перед ним не должно указываться ключевое слово `static`). Если употребить в качестве имени команды имя такого выполняемого файла, командный интерпретатор создаст новый процесс (см. следующий подраздел) и запустит в нем указанную выполняемую программу начиная с вызова функции `main`.

Тело функции `main`, вообще говоря, может быть произвольным (для интерпретатора существенно только наличие входной точки в программу с именем `main`), но для того, чтобы создать команду, которой можно задавать параметры, нужно придерживаться некоторых стандартных правил. В этом случае каждая функция `main` должна определяться с двумя параметрами - `argc` и `argv`. После вызова команды параметру `argc` будет соответствовать число символьных строк, указанных в качестве аргументов вызова команды, а `argv` - массив указателей на переменные, содержащие эти строки. При этом имя самой команды составляет первую строку аргументов (т.е. после вызова значение `argc` всегда больше или равно 1). Код функции `main` должен проанализировать допустимость заданного значения `argc` и соответствующим образом обработать заданные текстовые строки.

Например, следующий текст на языке Си может быть использован для создании команды, которая выводит на экран текстовую строку, заданную в качестве ее аргумента:

```
#include <stdio.h>

main (argc, argv)

int argc;

char *argv[];

{

if (argc != 2)

{ printf("usage: %s your-text\n", argv[0]);

exit;

}

printf("%s\n", argv[1]);

}
```

Процессы

Процесс в ОС UNIX - это программа, выполняемая в собственном виртуальном адресном пространстве. Когда пользователь входит в систему, автоматически создается процесс, в

котором выполняется программа командного интерпретатора. Если командному интерпретатору встречается команда, соответствующая выполняемому файлу, то он создает новый процесс и запускает в нем соответствующую программу, начиная с функции `main`. Эта запущенная программа, в свою очередь, может создать процесс и запустить в нем другую программу (она тоже должна содержать функцию `main`) и т.д.

Управление процессами подробно обсуждается в третьей части курса. Тем не менее кратко опишем здесь общий подход. Для образования нового процесса и запуска в нем программы используются два системных вызова (примитива ядра ОС UNIX) - `fork()` и `exec` (имя-выполняемого-файла). Системный вызов `fork` приводит к созданию нового адресного пространства, состояние которого абсолютно идентично состоянию адресного пространства основного процесса (т.е. в нем содержатся те же программы и данные).

Другими словами, сразу после выполнения системного вызова `fork` основной и порожденный процессы являются абсолютными близнецами; управление и в том, и в другом находится в точке, непосредственно следующей за вызовом `fork`. Чтобы программа могла разобраться, в каком процессе она теперь работает - в основном или порожденном, функция `fork` возвращает разные значения: 0 в порожденном процессе и целое положительное число (идентификатор порожденного процесса) в основном процессе.

Теперь, если мы хотим запустить новую программу в порожденном процессе, нужно обратиться к системному вызову `exec`, указав в качестве аргументов вызова имя файла, содержащего новую выполняемую программу, и, возможно, одну или несколько текстовых строк, которые будут переданы в качестве аргументов функции `main` новой программы. Выполнение системного вызова `exec` приводит к тому, что в адресное пространство порожденного процесса загружается новая выполняемая программа и запускается с адреса, соответствующего входу в функцию `main`.

В следующем примере пользовательская программа, вызываемая как команда `shell`, выполняет в отдельном процессе стандартную команду `shell ls`, которая выдает на экран содержимое текущего каталога файлов.

```
main()

{if(fork()==0) wait(0); /* родительский процесс */

else execl("ls", "ls", 0); /* порожденный процесс */

}
```

Перенаправление ввода/вывода

Механизм перенаправления ввода/вывода является одним из наиболее элегантных, мощных и одновременно простых механизмов ОС UNIX. Цель, которая ставилась при разработке этого механизма, состоит в следующем. Поскольку UNIX - это интерактивная система, то обычно программы вводят текстовые строки с терминала и выводят результирующие текстовые строки на экран терминала. Для того, чтобы обеспечить более гибкое использование таких программ, желательно уметь обеспечить им ввод из файла или из вывода других программ и направить их вывод в файл или на ввод другим программам.

Реализация механизма основывается на следующих свойствах ОС UNIX. Во-первых, любой ввод/вывод трактуется как ввод из некоторого файла и вывод в некоторый файл. Клавиатура и экран терминала тоже интерпретируются как файлы (первый можно только читать, а во второй можно только писать). Во-вторых, доступ к любому файлу производится через его дескриптор (положительное целое число). Фиксируются три значения дескрипторов файлов. Файл с дескриптором 1 называется файлом стандартного ввода (`stdin`), файл с дескриптором 2 - файлом стандартного вывода (`stdout`), и файл с дескриптором 3 - файлом стандартного вывода диагностических сообщений (`stderr`). В-третьих, программа, запущенная в некотором процессе, "наследует" от породившего процесса все дескрипторы открытых файлов.

В головном процессе интерпретатора командного языка файлом стандартного ввода является клавиатура терминала пользователя, а файлами стандартного вывода и вывода диагностических сообщений - экран терминала. Однако при запуске любой команды можно сообщить интерпретатору (средствами соответствующего командного языка), какой файл или вывод какой программы должен служить файлом стандартного ввода для запускаемой программы и какой файл или ввод какой программы должен служить файлом стандартного вывода или вывода диагностических сообщений для запускаемой программы. Тогда интерпретатор перед выполнением системного вызова `exec` открывает указанные файлы, подменяя смысл дескрипторов 1, 2 и 3.

Конечно, то же самое может проделать и любая другая программа, запускающая третью программу в специально созданном процессе. Следовательно, все, что требуется для нормального функционирования механизма перенаправления ввода/вывода - это придерживаться при программировании соглашения об использовании дескрипторов `stdin`, `stdout` и `stderr`. Это не очень трудно, поскольку в наиболее распространенных функциях библиотеки ввода/вывода `printf`, `scanf` и `error` вообще не требуется указывать дескриптор файла. Функция `printf` неявно использует `stdout`, функция `scanf` - `stdin`, а функция `error` - `stderr`.

Более подробно механизм перенаправления вывода одной программы на ввод другой программы будет рассмотрен в третьей части курса.

Ядро ОС UNIX

Как и в любой другой многопользовательской операционной системе, обеспечивающей защиту пользователей друг от друга и защиту системных данных от любого непривилегированного пользователя, в ОС UNIX имеется защищенное ядро, которое управляет ресурсами компьютера и предоставляет пользователям базовый набор услуг.

Следует заметить, что удобство и эффективность современных вариантов ОС UNIX не означает, что вся система, включая ядро, спроектирована и структуризована наилучшим образом. Как мы показали в первой части курса, ОС UNIX развивалась на протяжении многих лет (это первая в истории операционная система, которая продолжает завоевывать популярность в таком зрелом возрасте - уже больше 25 лет). Естественно, наращивались возможности системы, и, как это часто бывает в больших системах, качественные улучшения структуры ОС UNIX не поспевали за ростом ее возможностей.

В результате, ядро большинства современных коммерческих вариантов ОС UNIX (как мы отмечали ранее, почти все они основаны на UNIX System V) представляет собой не очень четко структуризованный монолит большого размера. По этой причине программирование на уровне ядра ОС UNIX продолжает оставаться искусством (если не

считать отработанной и понятной технологии разработки драйверов внешних устройств). Эта недостаточная технологичность организации ядра ОС UNIX многих не удовлетворяет. Отсюда стремление к полному воспроизведению среды ОС UNIX при полностью иной организации системы (в частности, с применением микроядерного подхода, который мы кратко рассмотрим в конце курса).

По причине наибольшей распространенности в этом подразделе мы в основном обсуждаем ядро UNIX System V (можно считать его традиционным). В конце курса мы обсудим отличия в организации ядра других ветвей иерархии вариантов ОС UNIX.

Общая организация традиционного ядра ОС UNIX

Одно из основных достижений ОС UNIX состоит в том, что система обладает свойством высокой мобильности. Смысл этого качества состоит в том, что вся операционная система, включая ее ядро, сравнительно просто переносится на различные аппаратные платформы. Все части системы, не считая ядра, являются полностью машинно-независимыми. Эти компоненты аккуратно написаны на языке Си, и для их переноса на новую платформу (по крайней мере, в классе 32-разрядных компьютеров) требуется только перекомпиляция исходных текстов в коды целевого компьютера.

Конечно, наибольшие проблемы связаны с ядром системы, которое полностью скрывает специфику используемого компьютера, но само зависит от этой специфики. В результате продуманного разделения машинно-зависимых и машинно-независимых компонентов ядра (видимо, с точки зрения разработчиков операционных систем, в этом состоит наивысшее достижение разработчиков традиционного ядра ОС UNIX) удалось добиться того, что основная часть ядра не зависит от архитектурных особенностей целевой платформы, написана полностью на языке Си и для переноса на новую платформу нуждается только в перекомпиляции.

Однако сравнительно небольшая часть ядра является машинно-зависимой и написана на смеси языка Си и языка ассемблера целевого процессора. При переносе системы на новую платформу требуется переписывание этой части ядра с использованием языка ассемблера и учетом специфических черт целевой аппаратуры. Машинно-зависимые части ядра хорошо изолированы от основной машинно-независимой части, и при хорошем понимании назначения каждого машинно-зависимого компонента переписывание машинно-зависимой части является в основном технической задачей (хотя и требует высокой программистской квалификации).

Машинно-зависимая часть традиционного ядра ОС UNIX включает следующие компоненты:

- раскрутка и инициализация системы на низком уровне (пока это зависит от особенностей аппаратуры);
- первичная обработка внутренних и внешних прерываний;
- управление памятью (в той части, которая относится к особенностям аппаратной поддержки виртуальной памяти);
- переключение контекста процессов между режимами пользователя и ядра;
- связанные с особенностями целевой платформы части драйверов устройств.

Основные функции

К основным функциям ядра ОС UNIX принято относить следующие:

(a) Инициализация системы - функция запуска и раскрутки. Ядро системы обеспечивает средство раскрутки (`bootstrap`), которое обеспечивает загрузку полного ядра в память компьютера и запускает ядро.

(b) Управление процессами и нитями - функция создания, завершения и отслеживания существующих процессов и нитей ("процессов", выполняемых на общей виртуальной памяти). Поскольку ОС UNIX является мультипроцессной операционной системой, ядро обеспечивает разделение между запущенными процессами времени процессора (или процессоров в мультипроцессорных системах) и других ресурсов компьютера для создания внешнего ощущения того, что процессы реально выполняются в параллель.

(c) Управление памятью - функция отображения практически неограниченной виртуальной памяти процессов в физическую оперативную память компьютера, которая имеет ограниченные размеры. Соответствующий компонент ядра обеспечивает разделяемое использование одних и тех же областей оперативной памяти несколькими процессами с использованием внешней памяти.

(d) Управление файлами - функция, реализующая абстракцию файловой системы, - иерархии каталогов и файлов. Файловые системы ОС UNIX поддерживают несколько типов файлов. Некоторые файлы могут содержать данные в формате ASCII, другие будут соответствовать внешним устройствам. В файловой системе хранятся объектные файлы, выполняемые файлы и т.д. Файлы обычно хранятся на устройствах внешней памяти; доступ к ним обеспечивается средствами ядра. В мире UNIX существует несколько типов организации файловых систем. Современные варианты ОС UNIX одновременно поддерживают большинство типов файловых систем.

(e) Коммуникационные средства - функция, обеспечивающая возможности обмена данными между процессами, выполняющимися внутри одного компьютера (IPC - Inter-Process Communications), между процессами, выполняющимися в разных узлах локальной или глобальной сети передачи данных, а также между процессами и драйверами внешних устройств.

(f) Программный интерфейс - функция, обеспечивающая доступ к возможностям ядра со стороны пользовательских процессов на основе механизма системных вызовов, оформленных в виде библиотеки функций.

В следующих разделах этой части курса и, более подробно, в третьей части курса мы будем знакомиться с базовыми возможностями ядра ОС UNIX.

Принципы взаимодействия с ядром

В любой операционной системе поддерживается некоторый механизм, который позволяет пользовательским программам обращаться за услугами 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 называются системными вызовами.

Название не изменяет смысл, который состоит в том, что для обращения к функциям ядра ОС используются "специальные команды" процессора, при выполнении которых возникает особого рода внутреннее прерывание процессора, переводящее его в режим ядра (в большинстве современных ОС этот вид прерываний называется `trap` - ловушка).

При обработке таких прерываний (дешифрации) ядро ОС распознает, что на самом деле прерывание является запросом к ядру со стороны пользовательской программы на выполнение определенных действий, выбирает параметры обращения и обрабатывает его, после чего выполняет "возврат из прерывания", возобновляя нормальное выполнение пользовательской программы.

Понятно, что конкретные механизмы возбуждения внутренних прерываний по инициативе пользовательской программы различаются в разных аппаратных архитектурах. Поскольку ОС UNIX стремится обеспечить среду, в которой пользовательские программы могли бы быть полностью мобильны, потребовался дополнительный уровень, скрывающий особенности конкретного механизма возбуждения внутренних прерываний. Этот механизм обеспечивается так называемой библиотекой системных вызовов.

Для пользователя библиотека системных вызовов представляет собой обычную библиотеку заранее реализованных функций системы программирования языка Си. При программировании на языке Си использование любой функции из библиотеки системных вызовов ничем не отличается от использования любой собственной или библиотечной Си-функции. Однако внутри любой функции конкретной библиотеки системных вызовов содержится код, являющийся, вообще говоря, специфичным для данной аппаратной платформы.

Наиболее важные системные вызовы ОС UNIX рассматриваются в оставшихся разделах этой части курса и в следующей части.

Принципы обработки прерываний

Конечно, применяемый в операционных системах механизм обработки внутренних и внешних прерываний в основном зависит от того, какая аппаратная поддержка обработки прерываний обеспечивается конкретной аппаратной платформой. К счастью, к настоящему моменту (и уже довольно давно) основные производители компьютеров де-факто пришли к соглашению о базовых механизмах прерываний.

Если говорить не очень точно и конкретно, суть принятого на сегодня механизма состоит в том, что каждому возможному прерыванию процессора (будь то внутреннее или внешнее прерывание) соответствует некоторый фиксированный адрес физической оперативной памяти. В тот момент, когда процессору разрешается прерваться по причине наличия внутренней или внешней заявки на прерывание, происходит аппаратная передача управления на ячейку физической оперативной памяти с соответствующим адресом - обычно адрес этой ячейки называется "вектором прерывания" (как правило, заявки на внутреннее прерывание, т.е. заявки, поступающие непосредственно от процессора, удовлетворяются немедленно).

Дело операционной системы - разместить в соответствующих ячейках оперативной памяти программный код, обеспечивающий начальную обработку прерывания и инициирующий полную обработку.

В основном, ОС UNIX придерживается общего подхода. В векторе прерывания, соответствующем внешнему прерыванию, т.е. прерыванию от некоторого внешнего устройства, содержатся команды, устанавливающие уровень выполнения процессора (уровень выполнения определяет, на какие внешние прерывания процессор должен реагировать незамедлительно) и осуществляющие переход на программу полной обработки прерывания в соответствующем драйвере устройства. Для внутреннего

прерывания (например, прерывания по инициативе программы пользователя при отсутствии в основной памяти нужной страницы виртуальной памяти, при возникновении исключительной ситуации в программе пользователя и т.д.) или прерывания от таймера в векторе прерывания содержится переход на соответствующую программу ядра ОС UNIX.

Файловая система

Как мы отмечали в разделе 2.1, понятие файла является одним из наиболее важных для ОС UNIX. Все файлы, с которыми могут манипулировать пользователи, располагаются в файловой системе, представляющей собой дерево, промежуточные вершины которого соответствуют каталогам, и листья - файлам и пустым каталогам. Примерная структура файловой системы ОС UNIX показана на рисунке 2.1. Реально на каждом логическом диске (разделе физического дискового пакета) располагается отдельная иерархия каталогов и файлов. Для получения общего дерева в динамике используется "монтирование" отдельных иерархий к фиксированной корневой файловой системе.

Замечание: в мире ОС UNIX по историческим причинам термин "файловая система" является перегруженным, обозначая одновременно иерархию каталогов и файлов и часть ядра, которая управляет каталогами и файлами. Видимо, было бы правильнее называть иерархию каталогов и файлов архивом файлов, а термин "файловая система" использовать только во втором смысле. Однако, следуя традиции ОС UNIX, мы будем использовать этот термин в двух смыслах, различая значения по контексту.

Каждый каталог и файл файловой системы имеет уникальное полное имя (в ОС UNIX это имя принято называть `full pathname` - имя, задающее полный путь, поскольку оно действительно задает полный путь от корня файловой системы через цепочку каталогов к соответствующему каталогу или файлу; мы будем использовать термин "полное имя", поскольку для pathname отсутствует благозвучный русский аналог). Каталог, являющийся корнем файловой системы (корневой каталог), в любой файловой системе имеет предопределенное имя "/" (слэш). Полное имя файла, например, `/bin/sh` означает, что в корневом каталоге должно содержаться имя каталога `bin`, а в каталоге `bin` должно содержаться имя файла `sh`. Коротким или относительным именем файла (`relative pathname`) называется имя (возможно, составное), задающее путь к файлу от текущего рабочего каталога (существует команда и соответствующий системный вызов, позволяющие установить текущий рабочий каталог).

В каждом каталоге содержатся два специальных имени, имя ".", именующее сам этот каталог, и имя "..", именующее "родительский" каталог данного каталога, т.е. каталог, непосредственно предшествующий данному в иерархии каталогов.

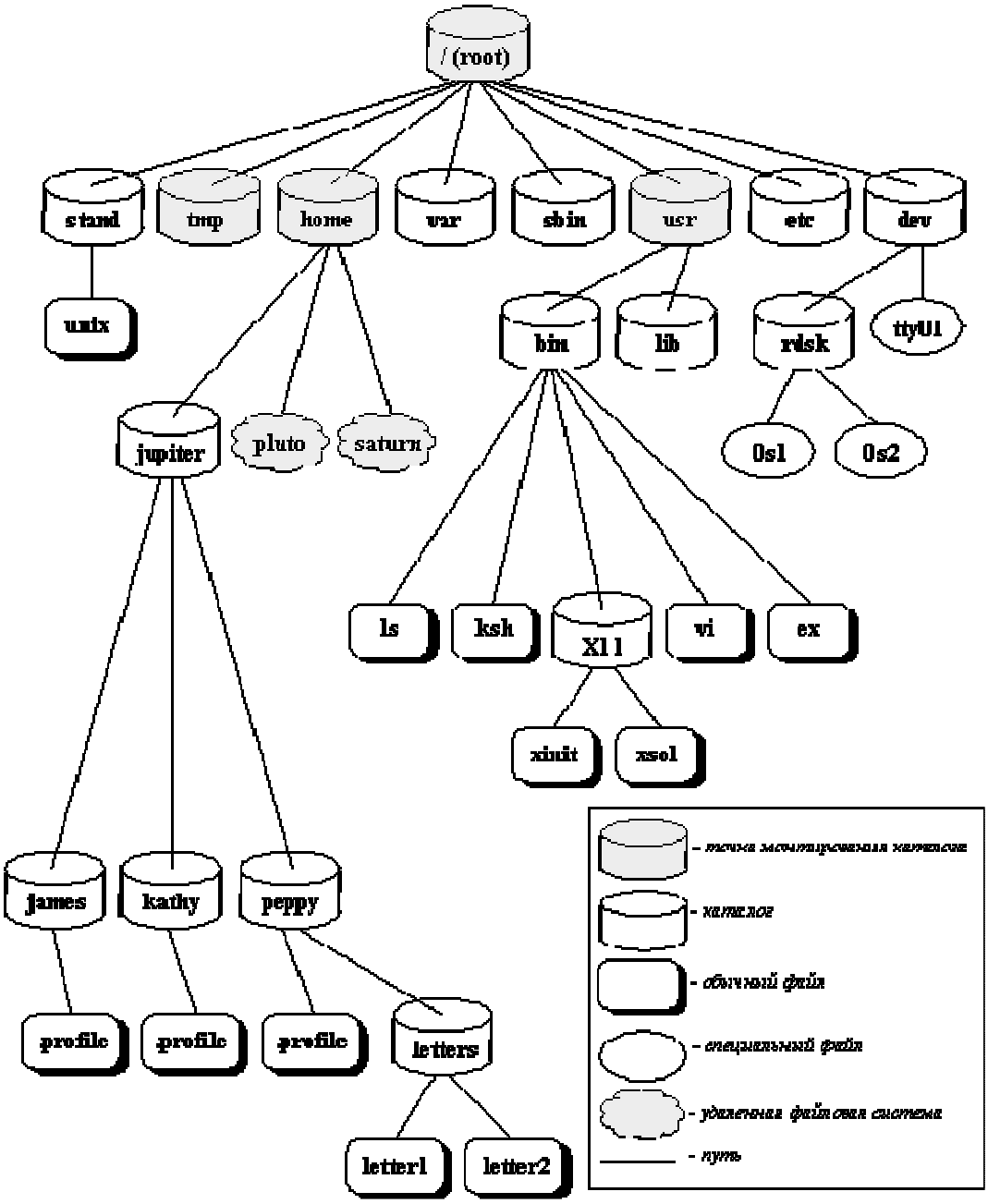

Рис. 2.1. Структура каталогов файловой системы

UNIX поддерживает многочисленные утилиты, позволяющие работать с файловой системой и доступные как команды командного интерпретатора. Вот некоторые из них (наиболее употребительные):

`cp имя1 имя2` - копирование файла `имя1` в файл `имя2`
`rm имя1` - уничтожение файла `имя1`
`mv имя1 имя2` - переименование файла `имя1` в файл `имя2`
`mkdir имя` - создание нового каталога `имя`
`rmdir имя` - уничтожение каталога `имя`
`ls имя` - выдача содержимого каталога `имя`
`cat имя` - выдача на экран содержимого файла `имя`
`chown имя режим` - изменение режима доступа к файлу

Структура файловой системы

Файловая система обычно размещается на дисках или других устройствах внешней памяти, имеющих блочную структуру. Кроме блоков, сохраняющих каталоги и файлы, во внешней памяти поддерживается еще несколько служебных областей.

В мире UNIX существует несколько разных видов файловых систем со своей структурой внешней памяти. Наиболее известны традиционная файловая система UNIX System V (`s5`) и файловая система семейства UNIX BSD (`ufs`). Файловая система `s5` состоит из четырех секций (рисунок 2.2,a). В файловой системе `ufs` на логическом диске (разделе реального диска) находится последовательность секций файловой системы (рисунок 2.2,b).

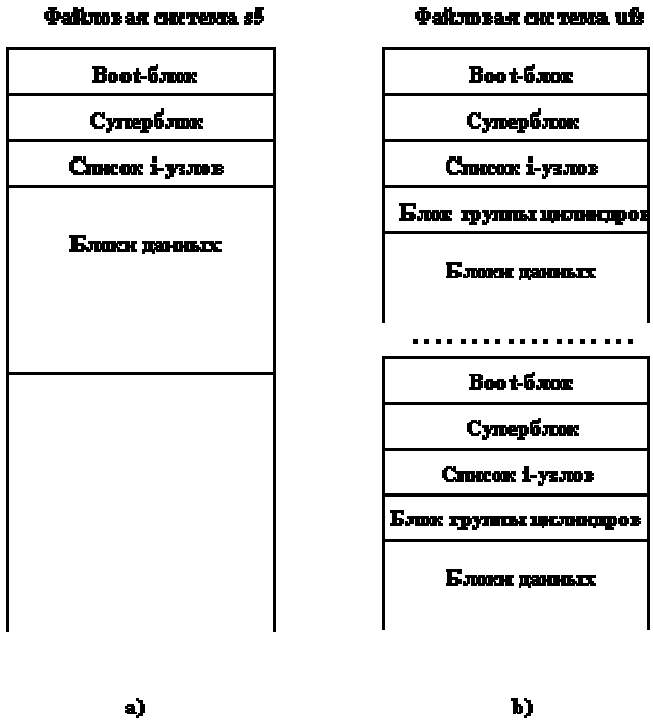

Рис. 2.2. Структура внешней памяти файловых систем `s5` и `ufs`

Кратко опишем суть и назначение каждой области диска.

- `Boot`-блок содержит программу раскрутки, которая служит для первоначального запуска ОС UNIX. В файловых системах `s5` реально используется `boot`-блок только корневой файловой системы. В дополнительных файловых системах эта область присутствует, но не используется.
- Суперблок - это наиболее ответственная область файловой системы, содержащая информацию, которая необходима для работы с файловой системой в целом. Суперблок содержит список свободных блоков и свободные i-узлы (information nodes - информационные узлы). В файловых системах `ufs` для повышения устойчивости поддерживается несколько копий суперблока (как видно из рисунка 2.2,b, по одной копии на группу цилиндров). Каждая копия суперблока имеет размер 8196 байт, и только одна копия суперблока используется при монтировании файловой системы (см. ниже). Однако, если при монтировании устанавливается, что первичная копия суперблока повреждена или не удовлетворяет критериям целостности информации, используется резервная копия.

- Блок группы цилиндров содержит число i-узлов, специфицированных в списке i-узлов для данной группы цилиндров, и число блоков данных, которые связаны с этими i-узлами. Размер блока группы цилиндров зависит от размера файловой системы. Для повышения эффективности файловая система `ufs` старается размещать i-узлы и блоки данных в одной и той же группе цилиндров.
- Список i-узлов (ilist) содержит список i-узлов, соответствующих файлам данной файловой системы. Максимальное число файлов, которые могут быть созданы в файловой системе, определяется числом доступных i-узлов. В i-узле хранится информация, описывающая файл: режимы доступа к файлу, время создания и последней модификации, идентификатор пользователя и идентификатор группы создателя файла, описание блочной структуры файла и т.д.
- Блоки данных - в этой части файловой системы хранятся реальные данные файлов. В случае файловой системы `ufs` все блоки данных одного файла пытаются разместить в одной группе цилиндров. Размер блока данных определяется при форматировании файловой системы командой `mkfs` и может быть установлен в 512, 1024, 2048, 4096 или 8192 байтов.

Монтируемые файловые системы

Файлы любой файловой системы становятся доступными только после "монтирования" этой файловой системы. Файлы "не смонтированной" файловой системы не являются видимыми операционной системой.

Для монтирования файловой системы используется системный вызов `mount`. Монтирование файловой системы означает следующее. В имеющемся к моменту монтирования дереве каталогов и файлов должен иметься листовой узел - пустой каталог (в терминологии UNIX такой каталог, используемый для монтирования файловой системы, называется `directory mount point` - точка монтирования). В любой файловой системе имеется корневой каталог. Во время выполнения системного вызова `mount` корневой каталог монтируемой файловой системы совмещается с каталогом - точкой монтирования, в результате чего образуется новая иерархия с полными именами каталогов и файлов.

Смонтированная файловая система впоследствии может быть отсоединена от общей иерархии с использованием системного вызова `umount`. Для успешного выполнения этого системного вызова требуется, чтобы отсоединяемая файловая система к этому моменту не находилась в использовании (т.е. ни один файл из этой файловой системы не был открыт). Корневая файловая система всегда является смонтированной, и к ней не применим системный вызов `umount`.

Как мы отмечали выше, отдельная файловая система обычно располагается на логическом диске, т.е. на разделе физического диска. Для инициализации файловой системы не поддерживаются какие-либо специальные системные вызовы. Новая файловая система образуется на отформатированном диске с использованием утилиты (команды) `mkfs`. Вновь созданная файловая система инициализируется в состояние, соответствующее наличию всего лишь одного пустого корневого каталога. Команда `mkfs` выполняет инициализацию путем прямой записи соответствующих данных на диск.

Интерфейс с файловой системой

Ядро ОС UNIX поддерживает для работы с файлами несколько системных вызовов. Среди них наиболее важными являются `open`, `creat`, `read`, `write`, `lseek` и `close`.

Важно отметить, 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 в внешними устройствами, при организации межпроцессных взаимодействий и т.д.

Файл в системных вызовах, обеспечивающих реальный доступ к данным, идентифицируется своим дескриптором (целым значением). Дескриптор файла выдается системными вызовами `open` (открыть файл) и `creat` (создать файл). Основным параметром операций открытия и создания файла является полное или относительное имя файла. Кроме того, при открытии файла указывается также режим открытия (только чтение, только запись, запись и чтение и т.д.) и характеристика, определяющая возможности доступа к файлу:

```
open(pathname, oflag [,mode])
```

Одним из признаков, которые могут участвовать в параметре `oflag`, является признак O_CREAT, наличие которого указывает на необходимость создания файла, если при выполнении системного вызова `open` файл с указанным именем не существует (параметр `mode` имеет смысл только при наличии этого признака). Тем не менее по историческим причинам и для обеспечения совместимости с предыдущими версиями ОС UNIX отдельно поддерживается системный вызов `creat`, выполняющий практически те же функции.

Открытый файл может использоваться для чтения и записи последовательностей байтов. Для этого поддерживаются два системных вызова:

```
read(fd, buffer, count) и write(fd, buffer, count)
```

Здесь `fd` - дескриптор файла (полученный при ранее выполненном системном вызове `open` или `creat`), `buffer` - указатель символьного массива и `count` - число байтов, которые должны быть прочитаны из файла или в него записаны. Значение функции `read` или `write` - целое число, которое совпадает со значением `count`, если операция заканчивается успешно, равно нулю при достижении конца файла и отрицательно при возникновении ошибок.

В каждом открытом файле существует текущая позиция. Сразу после открытия файл позиционируется на первый байт. Другими словами, если сразу после открытия файла выполняется системный вызов `read` (или `write`), то будут прочитаны (или записаны) первые `count` байтов содержимого файла (конечно, они будут успешно прочитаны только в том случае, если файл реально содержит по крайней мере `count` байтов). После выполнения системного вызова `read` (или `write`) указатель чтения/записи файла будет установлен в позицию `count+1` и т.д.

Такой, чисто последовательный стиль работы, оказывается во многих случаях достаточным, но часто бывает необходимо читать или изменять файл с произвольной позиции (например, как без такой возможности хранить в файле прямо индексируемые массивы данных?). Для явного позиционирования файла служит системный вызов

```
lseek(fd, offset, origin)
```

Как и раньше, здесь `fd` - дескриптор ранее открытого файла. Параметр `offset` задает значение относительного смещения указателя чтения/записи, а параметр `origin`

указывает, относительно какой позиции должно применяться смещение. Возможны три значения параметра `origin`. Значение 0 указывает, что значение `offset` должно рассматриваться как смещение относительно начала файла. Значение 1 означает, что значение `offset` является смещением относительно текущей позиции файла. Наконец, значение 2 говорит о том, что задается смещение относительно конца файла. Заметим, что типом данных параметра `offset` является `long int`. Это значит, что, во-первых, могут задаваться достаточно длинные смещения и, во-вторых, смещения могут быть положительными и отрицательными.

Например, после выполнения системного вызова

```
lseek(fd, 0, 0)
```

указатель чтения/записи соответствующего файла будет установлен на начало (на первый байт) файла. Системный вызов

```
lseek(fd, 0, 2)
```

установит указатель на конец файла. Наконец, выполнение системного вызова

```
lseek(fd, 10, 1)
```

приведет к увеличению текущего значения указателя на 10.

Естественно, системный вызов успешно завершается только в том случае, когда заново сформированное значение указателя не выходит за пределы существующих размеров файла.

Разновидности файлов

Как мы неоднократно отмечали, в ОС UNIX понятие файла является универсальной абстракцией, позволяющей работать с обычными файлами, содержащимися на устройствах внешней памяти; с устройствами, вообще говоря, отличающимися от устройств внешней памяти; с информацией, динамически генерируемой другими процессами и т.д. Для поддержки этих возможностей единообразным способом файловые системы ОС UNIX поддерживают несколько типов файлов, наиболее существенные из которых мы рассмотрим в этом разделе.

Обычные файлы

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 информацию. Файловая система не предписывает обычным файлам какую-либо структуру, обеспечивая на уровне пользователей представление обычного файла как последовательности байтов. Используя базовые системные вызовы (или функции библиотеки ввода/вывода, которые мы рассмотрим в разделе 4), пользователи могут как угодно структуризовать файлы. В частности, многие СУБД хранят базы данных в обычных файлах ОС UNIX.

Для некоторых файлов, которые должны интерпретироваться компонентами самой операционной системы, UNIX поддерживает фиксированную структуру. Наиболее

важным примером таких файлов являются объектные и выполняемые файлы. Структура этих файлов поддерживается компиляторами, редакторами связей и загрузчиком. Однако, эта структура неизвестна файловой системе. Для нее такие файлы по-прежнему являются обычными файлами.

Файлы-каталоги

Наличие обычных файлов недостаточно для организации иерархических файловых систем. Требуется наличие каталогов, которые сопоставляют имена файлов или каталогов с их физическим описанием. Каталоги представляют собой особый вид файлов, которые хранятся во внешней памяти подобно обычным файлам, но структура которых поддерживается самой файловой системой.

Структура файла-каталога очень проста. Фактически, каталог - это таблица, каждый элемент которой состоит из двух полей: номера i-узла данного файла в его файловой системе и имени файла, которое связано с этим номером (конечно, этот файл может быть и каталогом). Если просмотреть содержимое текущего рабочего каталога с помощью команды `ls -ai`, то можно получить, например, следующий вывод:

```
inode File

number name

_____________________________

33 .

122 ..

54 first_file

65 second_file

65 second_again

77 dir2
```

Этот вывод демонстрирует, что в любом каталоге содержатся два стандартных имени - "." и "..". Имени "." сопоставляется i-узел, соответствующий самому этому каталогу, а имени ".." - i-узел, соответствующий "родительскому" каталогу данного каталога. "Родительским" (parent) каталогом называется каталог, в котором содержится имя данного каталога. Файлы с именами `"first_file"` и `"second_file"` - это разные файлы с номерами i-узлов 54 и 65 соответственно. Файл `"second_again"` представляет пример так называемой жесткой ссылки: он имеет другое имя, но реально описывается тем же i-узлом, что и файл `"second_file"`. Наконец, последний элемент каталога описывает некоторый другой каталог с именем `"dir2"`.

Этот последний файл, как и любой обычный файл, хранится в файловой системе как набор блоков запоминающего устройства. Однако файловая система знает, что на самом деле это каталог со структурой, контролируемой файловой системой. Поэтому файлам-каталогам соответствует особый тип файла (обозначенный в их i-узлах), по отношению к которому возможно выполнение только специального набора системных вызовов:

`mkdir`, производящего новый каталог,

`rmdir`, удаляющий пустой (незаполненный) каталог,

`getdents`, позволяющего прочитать содержимое указанного каталога.

Отсутствует системный вызов, позволяющий прямо писать в файл-каталог. Какими бы правами вы не обладали по отношению к файлу-каталогу, прямая запись информации в него запрещена - прямое следствие фиксированной (и закрытой от пользователей) структуры файлов-каталогов. Запись в файлы-каталоги производится неявно при создании и уничтожении файлов и каталогов, однако читать из файла-каталога при наличии соответствующих прав можно (пример - стандартная утилита `ls`, которая как раз и пользуется системным вызовом `getdents`).

Специальные файлы

Специальные файлы не хранят данные. Они обеспечивают механизм отображения физических внешних устройств в имена файлов файловой системы. Каждому устройству, поддерживаемому системой, соответствует, по меньшей мере, один специальный файл. Специальные файлы создаются при выполнении системного вызова `mknod`, каждому специальному файлу соответствует порция программного обеспечения, называемая драйвером соответствующего устройства. При выполнении чтения или записи по отношению к специальному файлу, производится прямой вызов соответствующего драйвера, программный код которого отвечает за передачу данных между процессом пользователя и соответствующим физическим устройством.

При этом имена специальных файлов можно использовать практически всюду, где можно использовать имена обычных файлов. Например, команда

```
cp myfile /tmp/kuz
```

перепишет файл с именем `myfile` в подкаталог `kuz` рабочего каталога. В то же время, команда

```
cp myfile /dev/console
```

выдаст содержимое файла `myfile` на системную консоль вашей установки.

Различаются два типа специальных файлов - блочные и символьные (подробности см. в разделе 3.3). Блочные специальные файлы ассоциируются с такими внешними устройствами, обмен с которыми производится блоками байтов данных, размером 512, 1024, 4096 или 8192 байтов. Типичным примером подобных устройств являются магнитные диски. Файловые системы всегда находятся на блочных устройствах, так что в команде `mount` обязательно указывается некоторое блочное устройство.

Символьные специальные файлы ассоциируются с внешними устройствами, которые не обязательно требуют обмена блоками данных равного размера. Примерами таких устройств являются терминалы (в том числе, системная консоль), последовательные устройства, некоторые виды магнитных лент. Иногда символьные специальные файлы ассоциируются с магнитными дисками.

При обмене данными с блочным устройством система буферизует данные во внутреннем системном кеше. Через определенные интервалы времени система "выталкивает" буфера, при которых содержится метка "измененный". Кроме того, существуют системные вызовы `sync` и `fsync`, которые могут использоваться в пользовательских программах, и выполнение которых приводит к выталкиванию измененных буферов из общесистемного пула. Основная проблема состоит в том, что при аварийной остановке компьютера (например, при внезапном выключении электрического питания) содержимое системного кеша может быть утрачено. Тогда внешние блочные файлы могут оказаться в рассогласованном состоянии. Например, может быть не вытолкнут супер-блок файловой системы, хотя файловая система соответствует его вытолкнутому состоянию. Заметим, что в любом случае согласованное состояние файловой системы может быть восстановлено (конечно, не всегда без потерь пользовательской информации).

Обмены с символьными специальными файлами производятся напрямую, без использования системной буферизации.

Связывание файлов с разными именами

Файловая система ОС UNIX обеспечивает возможность связывания одного и того же файла с разными именами. Часто имеет смысл хранить под разными именами одну и ту же команду (выполняемый файл) командного интерпретатора. Например, выполняемый файл традиционного текстового редактора ОС UNIX `vi` обычно может вызываться под именами `ex`, `edit`, `vi`, `view` и `vedit`.

Можно узнать имена всех связей данного файла с помощью команды `ncheck`, если указать в числе ее параметров номер i-узла интересующего файла. Например, чтобы узнать все имена, под которыми возможен вызов редактора `vi`, можно выполнить следующую последовательность команд (третий аргумент команды `ncheck` представляет собой имя специального файла, ассоциированного с файловой системой `/usr`):

```
$ ls -i /usr/bin/vi

372 /usr/bin/vi

$ ncheck -i 372 /dev/dsk/sc0d0s5

/dev/dsk/sc0d0s5:

372 /usr/bin/edit

372 /usr/bin/ex

372 /usr/bin/vedit

372 /usr/bin/vi

372 /usr/bin/view
```

Ранее в большинстве версий ОС UNIX поддерживались только так называемые "жесткие" связи, означающие, что в соответствующем каталоге имени связи сопоставлялось имя i-узла соответствующего файла. Новые жесткие связи могут создаваться с помощью системного вызова `link`. При выполнении этого системного вызова создается новый элемент каталога с тем же номером i-узла, что и ранее существовавший файл.

Начиная с "быстрой файловой системы" университета Беркли, в мире UNIX появились "символические связи". Символическая связь создается с помощью системного вызова `symblink`. При выполнении этого системного вызова 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 создается отдельный i-узел и даже заводится отдельный блок данных для хранения потенциально длинного имени файла.

Для работы с символьными связями поддерживаются три специальных системных вызова:

`readlink` - читает имя файла, связанного с именуемой символической связью (это имя может соответствовать реальному файлу, специальному файлу, жесткой ссылке или вообще ничему); имя хранится в блоке данных, связанном с данной символической ссылкой;

`lstat` - аналогичен системному вызову `stat` (получить информацию о файле), но относится к символической ссылке;

`lchowm` - аналогичен системному вызову `chown`, но используется для смены пользователя и группы самой символической ссылки.

Именованные программные каналы

Программный канал (pipe) - это одно из наиболее традиционных средств межпроцессных взаимодействий в ОС UNIX. В русской терминологии использовались различные переводы слова pipe (начиная от "трубки" и заканчивая замечательным русским словом "пайп"). Мы считаем, что термин "программный канал" наиболее точно отражает смысл термина "pipe".

Основной принцип работы программного канала состоит в буферизации байтового вывода одного процесса и обеспечении возможности чтения содержимого программного канала другим процессом в режиме FIFO (т.е. первым будет прочитан байт, который раньше всего записан). В любом случае интерфейс программного канала совпадает с интерфейсом файла (т.е. используются те же самые системные вызовы `read` и `write`).

Однако различаются два подвида программных каналов - неименованные и именованные. Неименованные программные каналы появились на самой заре ОС UNIX (см. раздел 1.2). Неименованный программный канал создается процессом-предком, наследуется процессами-потомками, и обеспечивает тем самым возможность связи в иерархии порожденных процессов. Интерфейс неименованного программного канала совпадает с интерфейсом файла (более подробно см. п. 3.4.4). Однако, поскольку такие каналы не имеют имени, им не соответствует какой-либо элемент каталога в файловой системе.

Именованному программному каналу обязательно соответствует элемент некоторого каталога и даже собственный i-узел. Другими словами, именованный программный канал выглядит как обычный файл, но не содержащий никаких данных до тех пор, пока некоторый процесс не выполнит в него запись. После того, как некоторый другой процесс прочитает записанные в канал байты, этот файл снова становится пустым. В отличие от неименованных программных каналов, именованные программные каналы могут использоваться для связи любых процессов (т.е. не обязательно процессов, входящих в одну иерархию родства). Интерфейс именованного программного канала практически полностью совпадает с интерфейсом обычного файла (включая системные вызовы `open` и

close), хотя, конечно, необходимо учитывать, что поведение канала отличается от поведения файла (подробности см. в п. 3.4.4).

Файлы, отображаемые в виртуальную память

В современных версиях ОС UNIX (например, в UNIX System V.4) появилась возможность отображать обычные файлы в виртуальную память процесса с последующей работой с содержимым файла не с помощью системных вызовов read, write и lseek, а с помощью обычных операций чтения из памяти и записи в память. Заметим, что этот прием был базовым в историческом предшественнике ОС UNIX - операционной системе Multics (см. раздел 1.1).

Для отображения файла в виртуальную память, после открытия файла выполняется системный вызов mmap, действие которого состоит в том, что создается сегмент разделяемой памяти, ассоциированный с открытым файлом, и автоматически подключается к виртуальной памяти процесса (подробнее о разделяемой памяти см. п. 3.4.1). После этого процесс может читать из нового сегмента (реально будут читаться байты, содержащиеся в файле) и писать в него (реально все записи отображаются в файл). При закрытии файла соответствующий сегмент автоматически отключается от виртуальной памяти процесса и уничтожается, если только файл не подключен к виртуальной памяти некоторого другого процесса.

Несколько процессов могут одновременно открыть один и тот же файл и подключить его к своей виртуальной памяти системным вызовом mmap. Тогда любые изменения, производимые путем записи в соответствующий сегмент разделяемой памяти, будут сразу видны другим процессам.

Синхронизация при параллельном доступе к файлам

Исторически в ОС UNIX всегда 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.

В System V.4 появились средства, позволяющие процессам синхронизировать параллельный доступ к файлам. 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). Однако, по отношению к ОС UNIX такое решение принимать было слишком поздно, поскольку многочисленные созданные за время существования системы прикладные программы опирались на свойство отсутствия автоматической синхронизации.

Поэтому разработчикам пришлось пойти "обходным путем". Ядро ОС UNIX поддерживает дополнительный системный вызов fcntl, обеспечивающий такие вспомогательные функции, относящиеся к файловой системе, как получение информации о текущем режиме открытия файла, изменение текущего режима открытия и т.д. В System V.4 именно на системный вызов fcntl нагружены функции синхронизации.

С помощью этого системного вызова можно установить монопольную или совместную блокировку файла целиком или блокировать указанный диапазон байтов внутри файла. Допускаются 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 времени.

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.

Принципы защиты

Поскольку ОС UNIX с самого своего зарождения задумывалась как многопользовательская операционная система, в ней всегда была актуальна проблема авторизации доступа различных пользователей к файлам файловой системы. Под авторизацией доступа мы понимаем действия системы, которые допускают или не допускают доступ данного пользователя к данному файлу в зависимости от прав доступа пользователя и ограничений доступа, установленных для файла. Схема авторизации доступа, примененная в ОС UNIX, настолько проста и удобна и одновременно настолько мощна, что стала фактическим стандартом современных операционных систем (не претендующих на качества систем с многоуровневой защитой).

Идентификаторы пользователя и группы пользователей

С каждым выполняемым процессом в ОС UNIX связываются реальный идентификатор пользователя (`real user ID`), действующий идентификатор пользователя (`effective user ID`) и сохраненный идентификатор пользователя (`saved user ID`). Все эти идентификаторы устанавливаются с помощью системного вызова `setuid`, который можно выполнять только в режиме суперпользователя. Аналогично, с каждым процессом связываются три идентификатора группы пользователей - `real group ID`, `effective group ID` и `saved group ID`. Эти идентификаторы устанавливаются привилегированным системным вызовом `setgid`.

При входе пользователя в систему программа `login` проверяет, что пользователь зарегистрирован в системе и знает правильный пароль (если он установлен), образует новый процесс и запускает в нем требуемый для данного пользователя `shell`. Но перед этим `login` устанавливает для вновь созданного процесса идентификаторы пользователя и группы, используя для этого информацию, хранящуюся в файлах `/etc/passwd` и `/etc/group`. После того, как с процессом связаны идентификаторы пользователя и группы, для этого процесса начинают действовать ограничения для доступа к файлам. Процесс может получить доступ к файлу или выполнить его (если файл содержит выполняемую программу) только в том случае, если хранящиеся при файле ограничения доступа позволяют это сделать. Связанные с процессом идентификаторы передаются создаваемым им процессам, распространяя на них те же ограничения. Однако в некоторых случаях процесс может изменить свои права с помощью системных вызовов `setuid` и `setgid`, а иногда система может изменить права доступа процесса автоматически.

Рассмотрим, например, следующую ситуацию. В файл `/etc/passwd` запрещена запись всем, кроме суперпользователя (суперпользователь может писать в любой файл). Этот файл, помимо прочего, содержит пароли пользователей и каждому пользователю разрешается изменять свой пароль. Имеется специальная программа `/bin/passwd`, изменяющая пароли. Однако пользователь не может сделать это даже с помощью этой программы, поскольку запись в файл `/etc/passwd` запрещена. В системе UNIX эта проблема разрешается следующим образом. При выполняемом файле может быть указано, что при его запуске должны устанавливаться идентификаторы пользователя и/или группы. Если пользователь запрашивает выполнение такой программы (с помощью системного вызова `exec`), то для соответствующего процесса устанавливаются идентификатор пользователя, соответствующий идентификатору владельца выполняемого файла и/или идентификатор группы этого владельца. В частности, при запуске программы `/bin/passwd` процесс получит идентификатор суперпользователя, и программа сможет произвести запись в файл `/etc/passwd`.

И для идентификатора пользователя, и для идентификатора группы реальный ID является истинным идентификатором, а действующий ID - идентификатором текущего выполнения. Если текущий идентификатор пользователя соответствует суперпользователю, то этот идентификатор и идентификатор группы могут быть переустановлены в любое значение системными вызовами `setuid` и `setgid`. Если же текущий идентификатор пользователя отличается от идентификатора суперпользователя, то выполнение системных вызовов `setuid` и `setgid` приводит к замене текущего идентификатора истинным идентификатором (пользователя или группы соответственно).

Защита файлов

Как и принято в многопользовательской операционной системе, в UNIX поддерживается единообразный механизм контроля доступа к файлам и справочникам файловой системы. Любой процесс может получить доступ к некоторому файлу в том и только в том случае, если права доступа, описанные при файле, соответствуют возможностям данного процесса.

Защита файлов от несанкционированного доступа в ОС UNIX основывается на трех фактах. Во-первых, с любым процессом, создающим файл (или справочник), ассоциирован некоторый уникальный в системе идентификатор пользователя (UID - User Identifier), который в дальнейшем можно трактовать как идентификатор владельца вновь созданного файла. Во-вторых, с каждый процессом, пытающимся получить некоторый доступ к файлу, связана пара идентификаторов - текущие идентификаторы пользователя и его группы. В-третьих, каждому файлу однозначно соответствует его описатель - i-узел.

На последнем факте стоит остановиться более подробно. Важно понимать, что имена файлов и файлы как таковые - это не одно и то же. В частности, при наличии нескольких жестких связей с одним файлом несколько имен файла реально представляют один и тот же файл и ассоциированы с одним и тем же i-узлом. Любому используемому в файловой системе i-узлу всегда однозначно соответствует один и только один файл. I-узел содержит достаточно много разнообразной информации (большая ее часть доступна пользователям через системные вызовы `stat` и `fstat`), и среди этой информации находится часть, позволяющая файловой системе оценить правомощность доступа данного процесса к данному файлу в требуемом режиме.

Общие принципы защиты одинаковы для всех существующих вариантов системы: Информация i-узла включает UID и GID текущего владельца файла (немедленно после

создания файла идентификаторы его текущего владельца устанавливаются соответствующими действующим идентификатором процесса-создателя, но в дальнейшем могут быть изменены системными вызовами `chown` и `chgrp`). Кроме того, в i-узле файла хранится шкала, в которой отмечено, что может делать с файлом пользователь - его владелец, что могут делать с файлом пользователи, входящие в ту же группу пользователей, что и владелец, и что могут делать с файлом остальные пользователи. Мелкие детали реализации в разных вариантах системы различаются. Для определенности мы приведем точную картину того, как это происходит в UNIX System V Release 4 (таблица 2.1).

Таблица 2.1.
Представление информации, ограничивающей доступ к файлу, в i-узле файла

Шкала ограничений в восьмеричном виде	**Описание**
04000	Устанавливать идентификатор пользователя-владельца при выполнении файла.
020n0	При n = 7, 5, 3 или 1 устанавливать идентификатор группы владельца при выполнении файла. При n = 6, 4, 2 или 0 разрешается блокирование диапазонов адресов файла.
01000	Сохранять в области подкачки образ кодового сегмента выполняемого файла после конца его выполнения.
00400	Владельцу файла разрешено чтение файла.
00200	Владелец файла может дополнять или модифицировать файл.
00100	Владелец файла может его исполнять, если файл - исполняемый, или производить в нем поиск, если это файл-каталог.
00040	Все пользователи группы владельца могут читать файл.
00020	Все пользователи группы владельца могут дополнять или модифицировать файл.
00010	Все пользователи группы владельца могут исполнять файл, если файл - исполняемый, или производить в нем поиск, если это файл-каталог.
00004	Все пользователи могут читать файл.
00002	Все пользователи могут дополнять или модифицировать файл.
00001	Все пользователи могут исполнять файл, если файл - исполняемый, или производить в нем поиск, если это файл-каталог.

Управление устройствами

Управление внешними устройствами - это одна из важнейших функций любой операционной системы. Система должна обеспечивать эффективный и удобный доступ к периферийным устройствам, а также обеспечивать возможность унифицированной разработки программного обеспечения для вновь подключаемых внешних устройств. Рассмотрим, как эта проблема решается в ОС UNIX.

Устройство как специальный файл

В п. 2.4.4 мы уже отмечали, что для доступа к внешним устройствам в ОС UNIX используется универсальная абстракция файла. Помимо настоящих файлов (обычных файлов или каталогов), которые реально занимают память на магнитных дисках, файловая система содержит так называемые специальные файлы, для которых, как и для настоящих файлов, отводятся отдельные i-узлы, но которым на самом деле соответствуют внешние устройства. Это решение позволяет естественным образом работать в одном и том же интерфейсе с любым файлом или внешним устройством. (На самом деле, в некоторых случаях нестандартных внешних устройств приходится выходить за пределы стандартного интерфейса. Подробности и детали приводятся в разделе 3.3.).

Драйверы устройств

Любому программисту должно быть ясно, что простое объявление внешнего устройства специальным файлом не даст возможности работать с этим устройством, если не создан и соответствующим образом не подключен к системе специальный программный код, соответствующий специфике данного устройства. Как и в большинстве современных операционных систем, такого рода программный код в ОС UNIX называется драйвером устройства (в этом контексте слово драйвер лучше всего понимать в значении "управляющий").

Для профессионалов в области операционных систем драйверы ОС UNIX, в сущности, не представляют ничего нового. По-простому говоря, в любой системе драйвер устройства - это многовходовой программный модуль со своими статическими данными, который умеет инициировать работу с устройством, выполнять заказываемые пользователем обмены (на ввод или вывод данных), терминировать работу с устройством и обрабатывать прерывания от устройства. Однако, в любой операционной системе имеется своя технология разработки драйверов. В частности, в ОС UNIX различаются символьные, блочные и потоковые драйверы.

Символьные драйверы являются простейшими в ОС UNIX и предназначаются для обслуживания устройств, которые реально ориентированы на прием или выдачу произвольных последовательностей байтов (например, простой принтер или устройство ввода с перфоленты). Такие драйверы используют минимальный набор стандартных функций ядра UNIX, которые главным образом заключаются в возможности взять данные из виртуального пространства пользовательского процесса и/или поместить данные в такое виртуальное пространство.

Блочные драйверы - более сложные. Они работают с использованием возможностей системной буферизации блочных обменов ядра ОС UNIX. В число функций такого драйвера входит включение соответствующего блока данных в систему буферов ядра ОС UNIX и/или взятие содержимого буферной области в случае необходимости.

Наконец, наиболее сложной организацией отличаются потоковые драйверы. Фактически, такой драйвер представляет собой конвейер модулей, обеспечивающий многоступенчатую обработку запросов пользователя. Потоковые драйверы в среде ОС UNIX в основном предназначены для реализации доступа к сетевым устройствам, которые должны работать в соответствии с многоуровневыми сетевыми протоколами.

Подробности по поводу разных способов организации драйверов в ОС UNIX см. в разделе 3.3.

Последний момент, на который мы хотим обратить внимание в этом пункте, состоит в том, что (опять же, как и в большинстве развитых операционных систем) в ОС UNIX возможны два способа включения драйвера в состав ядра ОС. Первый способ состоит в полном включении драйвера в состав ядра на стадии генерации системы (т.е. драйвер статически объявляется частью ядра системы). Второй способ позволяет обойтись минимальным количеством статических объявлений на стадии генерации ядра (фактически, обеспечиваются лишь необходимые элементы статических таблиц). В любой момент работы системы такой драйвер может быть динамически загружен в ядро системы. После появления (статического или динамического) в ядре ОС UNIX драйверы всех разновидностей функционируют единообразно.

Внешний и внутренний интерфейсы устройств

Независимо от типа файла (обычный файл, каталог, связь или специальный файл) пользовательский процесс может работать с файлом через стандартный интерфейс, включающий системные вызовы `open`, `close`, `read` и `write`. Ядро само распознает, нужно ли обратиться к его стандартным функциям или вызвать подпрограмму драйвера устройства. Другими словами, если процесс пользователя открывает для чтения обычный файл, то системные вызовы `open` и `read` обрабатываются встроенными в ядро подпрограммами `open` и `read` соответственно. Однако, если файл является специальным, то будут вызваны подпрограммы `open` и `read`, определенные в соответствующем драйвере устройства (рисунок 2.3).

Кратко поясним этот рисунок. С каждым специальным файлом в системе связаны старший (`major`) и младший (`minor`) номера. После того, как (по содержанию i-узла) файловая система распознает, что данный файл является специальным, ядро ОС UNIX использует старший номер специального файла как индекс в конфигурационной таблице драйверов устройств. Поддерживаются две раздельные таблицы для символьных и блочных специальных файлов (или соответствующих драйверов). Для блочных драйверов используется системная таблица `bdevsw`, а для символьных - `cdevsw`. В обоих случаях элементом таблицы является структура (в терминах языка программирования Си), элементы которой содержат указатели на подпрограммы соответствующего драйвера. Допускается (и на самом деле используется) реализация драйверов, которые одновременно могут обрабатывать и блочный, и символьный ввод/вывод. В этом случае для драйвера будут существовать и элемент таблицы `bdevsw`, и таблицы `cdevsw`.

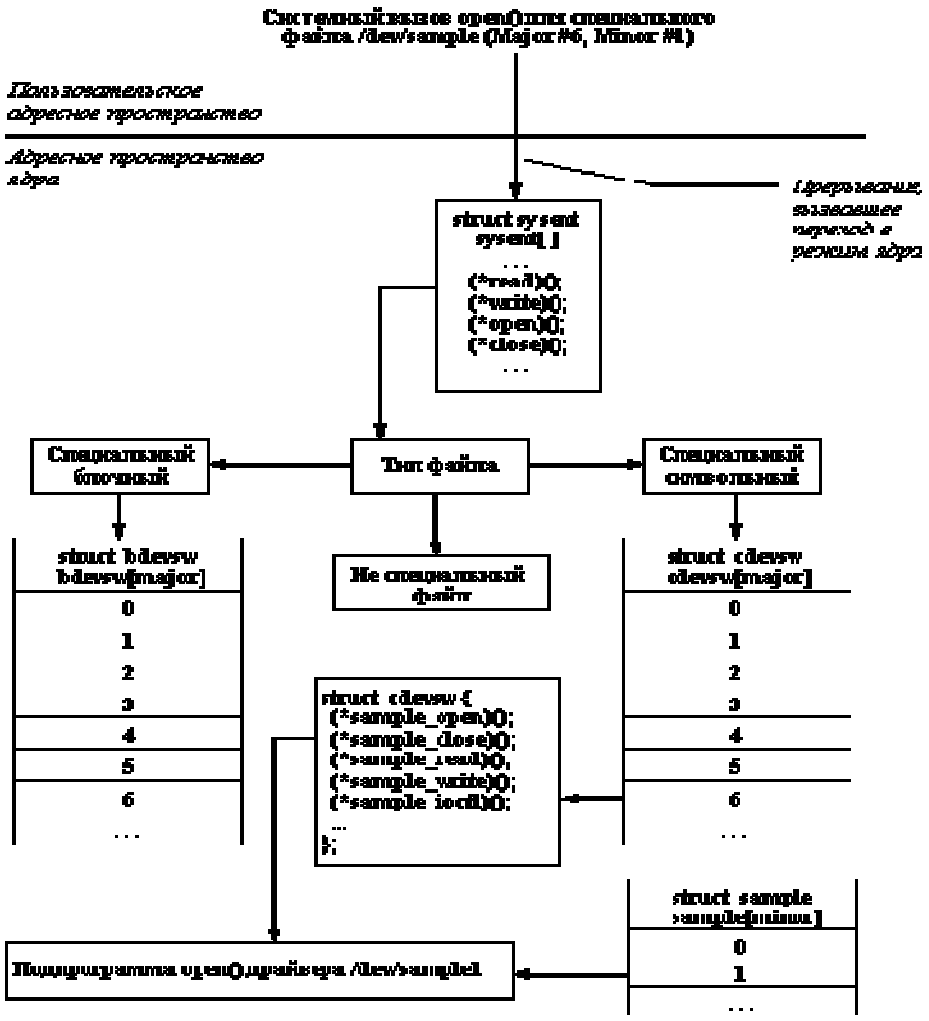

Рис. 2.3. Логическое представление открытия специального символьного файла

Старшему номеру специального файла блочного или специального файла, вообще говоря, соответствуют разные драйверы. Например, символьному специальному файлу /dev/tty и блочному специальному файлу /dev/swap в UNIX System V соответствует старший номер 6. Но поскольку первый специальный файл - символьный, а второй - блочный, они могут использовать один и тот же старший номер, хотя им соответствуют разные драйверы. В любом случае, младший номер специального файла передается в качестве параметра соответствующей функции драйвера, который волен использовать его любым образом, хотя обычно младший номер используется в качестве номера устройства, обслуживаемого аппаратным контроллером, которым на самом деле управляет данный драйвер. Другими словами, один драйвер как программная единица может управлять несколькими физическими устройствами.

Базовые механизмы сетевых взаимодействий

Операционная система UNIX с самого своего возникновения была по своей сути сетевой операционной системой. Однако по причине одновременного наличия нескольких вариантов ОС (см. раздел 1) образовалось несколько альтернативных механизмов, каждый из которых обладал собственными преимуществами и недостатками. В наиболее унифицированном и стандартизированном варианте UNIX System V среди этих механизмов был наведен некоторый порядок, и в этом разделе мы приведем сравнительно краткий обзор современного положения дел.

Потоки (Streams)

В самых ранних вариантах UNIX коммуникационные средства основывались на символьном вводе/выводе, главным образом потому, что аппаратной основой являлись модемы и терминалы. Поскольку такие устройства являются относительно медленными, в ранних вариантах не требовалось особенно заботиться о модульности и эффективности программного обеспечения. Несколько позже в системе появилась поддержка более развитых устройств, протоколов, операционных режимов и т.д., но программные средства по-прежнему основывались на ограниченных возможностях символьного ввода/вывода.

С появлением многоуровневых сетевых протоколов, таких как TCP/IP (US Defense Advanced Research Project Agency's Transmission Control Protocol/Internet Protocol), SNA (IBM's System Network Architecture), OSI (Open Systems Internetworking), X.25 и др. стало понятно, что в ОС UNIX требуется некоторая общая основа организации сетевых средств, основанных на многоуровневых протоколах. Для решения этой проблемы было реализовано несколько механизмов, обладающих примерно одинаковыми возможностями, но не совместимых между собой, поскольку каждый из них являлся результатом некоторого индивидуального проекта.

Общей проблемой ОС UNIX было то, что слабая развитость подсистемы ввода/вывода требовала решения задачи проектирования и включения в систему нового драйвера при каждом подключении нового устройства. Хотя зачастую уже существовал программный код, обладающий хотя бы частью функций, требуемых в новом драйвере, отсутствовала возможность использования этого существующего кода.

Во многом эта проблема была решена компанией AT&T, которая предложила и реализовала механизм потоков (STREAMS), обеспечивающий гибкие и модульные возможности для реализации драйверов устройств и коммуникационных протоколов. Потоки были впервые реализованы Деннисом Ритчи в 1984 году и были включены в пакет Networking Support Facilities (NSU) UNIX System V Release 3.

В UNIX System V Release 3 потоки были включены как основа реализации существующего символьного ввода/вывода. Однако в Release 4 в реализацию потоков были включены интерфейс драйвера устройства (DDI - Device Driver Interface) и интерфейс между драйвером и ядром (DKI - Device Kernel Interface), которые в совокупности одновременно обеспечивают возможности по взаимодействию драйвера устройства с ядром системы и простоту повторного использования имеющегося исходного кода драйверов. С использованием механизма потоков были переписаны практически все символьные драйверы, полностью переработаны подсистема управления терминалами и механизм программных каналов (pipes).

Если не вдаваться в детали, Streams представляют собой связанный набор средств общего назначения, включающий системные вызовы и подпрограммы, а также ресурсы ядра. В совокупности эти средства обеспечивают стандартный интерфейс символьного ввода/вывода внутри ядра, а также между ядром и соответствующими драйверами устройств, предоставляя гибкие и развитые возможности разработки и реализации коммуникационных сервисов. При этом механизм потоков не навязывает какой-либо конкретной архитектуры сети и/или конкретных протоколов. Как и любой другой драйвер устройства, потоковый драйвер представляется специальным файлом файловой системы со стандартным набором операций: `open`, `close`, `read`, `write` и `ioctl`. Простейшая форма организации потокового интерфейса показана на рисунке 2.4.

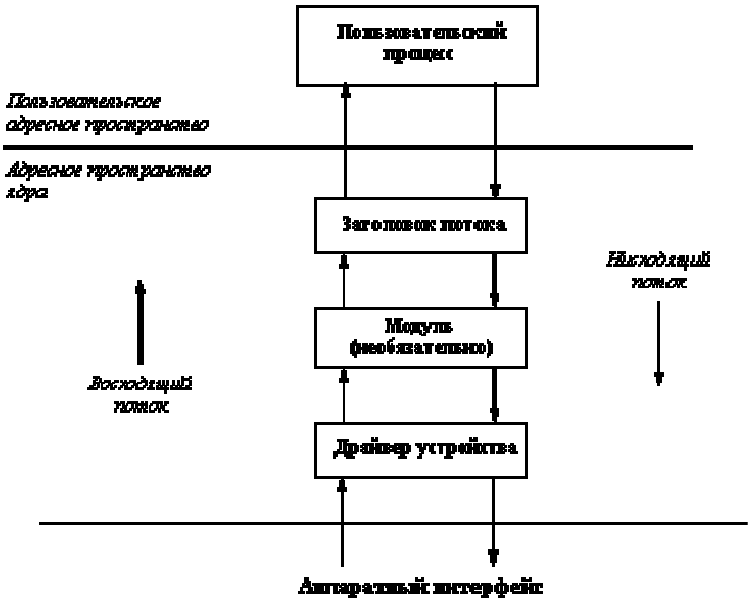

Рис. 2.4. Простая форма потокового интерфейса

Когда пользовательский процесс открывает потоковое устройство, пользуясь системным вызовом `open`, ядро связывает с драйвером заголовок потока. После этого пользовательский процесс общается с заголовком потока так, как если бы он представлял собой обычный драйвер устройства. Другими словами, заголовок потока отвечает за обработку всех системных вызовов, производимых пользовательским процессом по отношению к потоковому драйверу. Если процесс выполняет запись в устройство (системный вызов `write`), заголовок потока передает данные драйверу устройства в нисходящем направлении. Аналогично, при реализации чтения из устройства (системный вызов `read`) драйвер устройства передает данные заголовку потока в восходящем направлении.

В описанной схеме данные между заголовком потока и драйвером устройства передаются в неизменяемом виде без какой-либо промежуточной обработки. Однако можно добиться того, чтобы данные подвергались обработке при передаче их в любом направлении, если включить в поток между заголовком и драйвером устройства один или несколько потоковых модулей. Потоковый модуль является обработчиком данных, выполняющим определенный набор функций над данными по мере их прохождения по потоку. Простейшими примерами потокового модуля являются разного рода перекодировщики символьной информации. Более сложным примером является потоковый модуль, осуществляющий разборку нисходящих данных в пакеты для их передачи по сети и сборку восходящих данных с удалением служебной информации пакетов.

Каждый потоковый модуль является, вообще говоря, независимым от присутствия в потоке других модулей, обрабатывающих данные. Данные могут подвергаться обработке произвольным числом потоковых модулей, пока в конце концов не достигнут драйвера устройств при движении в нисходящем направлении или заголовка потока при движении в восходящем направлении. Для передачи данных от заголовка к драйверу или модулю, от одного модуля другому и от драйвера или модуля к заголовку потока используется механизм сообщений. Каждое сообщение представляет собой набор блоков сообщения, каждый из которых состоит из заголовка, блока данных и буфера данных.

Стек протоколов TCP/IP

TCP/IP (Transmission Control Protocol/Internet Protocol) представляет собой семейство протоколов, основным назначением которых является обеспечение возможности полезного сосуществования компьютерных сетей, основанных на разных технологиях. В 1969 году Агентство перспективных исследовательских проектов министерства обороны США (DARPA - Department of Defense Advanced Research Project Agency) поддержало и финансировало проект, посвященный поиску общей основы связи сетей с разной технологией. В результате выполнения этого проекта была образована единая виртуальная сеть, получившая название Internet. В Internet для связи независимых сетей, или доменов используется набор шлюзов. Каждый индивидуальный узел сети (Host) идентифицируется уникальным адресом, называемым адресом в Internet.

Для разрешения проблемы различий в форматах кадров, используемых в разных сетях, был определен универсальный формат пакета данных, называемого IP-датаграммой (Internet Protocol Datagram), состоящего из заголовка и порции данных и поэтому похожего на обычный сетевой кадр. Однако порция данных IP-датаграммы сама содержится внутри сетевого кадра, т.е. IP-датаграмма погружается в сетевой кадр конкретного формата и поэтому может передаваться в разных сетях, входящих в Internet. Все узлы, шлюзы и сети Internet должны быть в состоянии понимать IP-датаграммы.

Узлы, взаимодействующие в Internet, не устанавливают между собой физические соединения для целей индивидуального взаимодействия. Поэтому датаграммы не обрабатываются в каком-либо конкретном порядке. Напротив, каждая датаграмма обрабатывается независимо от других, что позволяет эффективно разделять ресурсы для всего множества (логически) связанных узлов. Но это, к сожалению, означает, что сервис, предоставляемый Internet, не является надежным, поскольку не гарантирует доставку пакетов в нужном порядке, отсутствие потерь датаграмм или отсутствие их дублирования.

Эту проблему решает протокол TCP (Transmission Control Protocol), обеспечивающий надежную доставку сообщений за счет подтверждений доставки датаграмм и их повторной передачи в случае надобности. Если узел, посылающий датаграмму, не получает подтверждение о ее доставке в течение установленного промежутка времени, то считается, что датаграмма не доставлена, и она посылается повторно.

Полное семейство протоколов, основанных на использовании IP-датаграмм, называется TCP/IP. Наиболее важными и базисными протоколами этого семейства (или стека, как его часто называют) являются кратко описанные выше протоколы IP и TCP. Мы не будем описывать остальные протоколы семейства TCP/IP. Для определенности все они перечислены в таблице 2.2. Большая часть коммуникационных средств ОС UNIX основывается на использовании протоколов стека TCP/IP.

Таблица 2.2.
Семейство протоколов TCP/IP

Название протокола	**Описание протокола**
TCP	Протокол управления передачей (Transmission Control Protocol)
UDP	Протокол пользовательских датаграмм (User Datagram Protocol)
ARP	Протокол разрешения адресов (Address Resolution Protocol)

RARP	Протокол обратного разрешения адресов (Reverse Address Resolution Protocol)
IP	Протокол Internet (Internet Protocol)
ICMP	Протокол управляющих сообщений Internet (Internet Control Message Protocol)
FTP	Протокол пересылки файлов (File Transfer Protocol)
TFTP	Простой протокол пересылки файлов (Trivial File Transfer Protocol)

В UNIX System V Release 4 протокол TCP/IP реализован как набор потоковых модулей плюс дополнительный компонент TLI (Transport Level Interface - Интерфейс транспортного уровня). TLI является интерфейсом между прикладной программой и транспортным механизмом. Приложение, пользующееся интерфейсом TLI, получает возможность использовать TCP/IP.

Интерфейс TLI основан на использовании классической семиуровневой модели ISO/OSI, которая разделяет сетевые функции на семь областей, или уровней. Цель модели в обеспечении стандарта сетевой связи компьютеров независимо от производителя аппаратуры компьютеров и/или сети. Семь уровней модели можно кратко описать следующим образом.

Уровень 1: Физический уровень (Physical Level) - среда передачи (например, Ethernet). Уровень отвечает за передачу неструктурированных данных по сети.

Уровень 2: Канальный уровень (Data Link Layer) - уровень драйвера устройства, называемый также уровнем ARP/RARP в TCP/IP. Этот уровень, в частности, отвечает за преобразование данных при исправлении ошибок, происходящих на физическом уровне.

Уровень 3: Сетевой уровень (Network Level) - отвечает за выполнение промежуточных сетевых функций, таких как поиск коммуникационного маршрута при отсутствии возможности прямой связи между узлом-отправителем и узлом-получателем. В TCP/IP этот уровень соответствует протоколам IP и ICMP.

Уровень 4: Транспортный уровень (Transport Level) - уровень протоколов TCP/IP или UDP/IP семейства протоколов TCP/IP. Уровень отвечает за разборку сообщения на фрагменты (пакеты) при передаче и за сборку полного сообщения из пакетов при приеме таким образом, что на более старших уровнях модели эти процедуры вообще незаметны. Кроме того, на этом уровне выполняется посылка и обработка подтверждений и, при необходимости, повторная передача.

Уровень 5: Уровень сессий (Session Layer) - отвечает за управление переговорами взаимодействующих транспортных уровней. В NFS (Network File System - Сетевая файловая система, см. п. 2.8.1) этот уровень используется для реализации механизма вызовов удаленных процедур (RPC - Remote Procedure Calls, см. п. 2.7.4).

Уровень 6: Уровень представлений (Presentation Layer) - отвечает за управление представлением информации. В NFS на этом уровне реализуется механизм внешнего представления данных (XDR - External Data Representation), машинно-независимого представления, понятного для всех компьютеров, входящих в сеть.

Уровень 7: Уровень приложений - интерфейс с такими сетевыми приложениями, как telnet, rlogin, mail и т.д.

Интерфейс TLI соответствует трем старшим уровням этой модели (с пятого по седьмой) и позволяет прикладному процессу пользоваться сервисами сети (без необходимости знать о деталях транспортного и более низких уровней). В System V Release 4 TLI реализован на основе механизма потоков. Для доступа используются не специальные системные вызовы, а функции библиотеки `/usr/lib/libnsl_s.a`.

Программные гнезда (Sockets)

Механизм программных гнезд (Sockets) впервые был реализован в 1982 году в UNIX BSD 4.1 в качестве развитого средства межпроцессных взаимодействий. Это средство, вообще говоря, позволяет любому процессу обмениваться сообщениями с любым другим процессом, независимо от того, выполняются они на одном компьютере или на разных, соединенных сетью. Функционально механизм программных гнезд близок к возможностям TLI (пятого уровня в соответствии с моделью ISO/OSI).

Программные гнезда входят в число обязательных компонентов стандартной среды ОС UNIX, однако реализуются в разных системах по-разному. В BSD-ориентированных системах Sockets исторически реализуются в ядре ОС, и пользователям предоставляются пять специальных системных вызовов: `socket`, `bind`, `listen`, `connect` и `accept` (подробнее о функциях этих системных вызовов см. п. 3.4.5).

В UNIX System V Release 4 тоже поддерживается механизм программных гнезд, однако он реализован не внутри ядра системы, а в виде набора библиотечных функций (библиотеки `/usr/lib/libsocket.a`), которые написаны с использованием механизма TLI. Заметим, что это в очередной раз демонстрирует преимущества подхода открытых систем, который всегда поддерживался в мире ОС UNIX: при наличии четко определенных интерфейсов и развитых базовых средств прикладной программист и разработанные им программы не должны зависеть от конкретной реализации.

Тем не менее, разработчики и поставщики System V призывают не использовать механизм Sockets в новых программах, а опираться непосредственно на возможности TLI. По нашему мнению, это дело вкуса, поскольку существует так много давно написанных программ, использующих программные гнезда, что ни один поставщик ОС UNIX никогда не решится перестать поддерживать Sockets.

Вызовы удаленных процедур (RPC)

Основными идеями механизма вызова удаленных процедур (RPC - Remote Procedure Calls) являются следующие:

(а) Во многих случаях взаимодействие процессов носит ярко выраженный асимметричный характер. Один из процессов ("клиент") запрашивает у другого процесса ("сервера") некоторую услугу (сервис) и не продолжает свое выполнение до тех пор, пока эта услуга не будет выполнена (и пока процесс-клиент не получит соответствующие результаты). Видно, что семантически такой режим взаимодействия эквивалентен вызову процедуры, и естественно желание оформить его должным образом синтаксически.

(б) Как уже отмечалось, ОС UNIX по своей идеологии с самого начала была по-настоящему сетевой операционной системой. Свойства переносимости позволяют, в частности, предельно просто создавать "операционно однородные" сети, включающие разнородные компьютеры. Однако, остается проблема разного представления данных в компьютерах разной архитектуры (часто по-разному представляются числа с плавающей

точкой, используется разный порядок размещения байтов в машинном слове и т.д.). Плохо, когда решение проблемы разных представлений данных возлагается на пользователей. Поэтому второй идеей RPC (многие считают, что это основная идея) является автоматическое обеспечение преобразования форматов данных при взаимодействии процессов, выполняющихся на разнородных компьютерах.

Впервые пакет RPC был реализован компанией Sun Microsystems в 1984 году в рамках ее продукта NFS (Network File System - сетевая файловая система, см. п. 2.8.1). Пакет был тщательно специфицирован с тем, чтобы пользовательский интерфейс и его функции не были зависимыми от применяемого транспортного механизма. Заметим, что в настоящее время Sun распространяет два варианта пакета - бесплатный (Public Domain), основанный на использовании программных гнезд, и коммерческий, базирующийся на механизме потоков (на самом деле, на интерфейсе TLI). В обоих случаях пакет реализуется как набор библиотечных функций. Например, в случае использования коммерческого варианта RPC в среде System V программы должны компоноваться с библиотекой `/usr/lib/librpcsvc.a`. Специальные системные вызовы для реализации RPC не поддерживаются.

Независимость от конкретного машинного представления данных обеспечивается отдельно специфицированным протоколом XDR (EXternal Data Representation - внешнее представление данных). Этот протокол определяет стандартный способ представления данных, скрывающий такие машинно-зависимые свойства, как порядок байтов в слове, требования к выравниванию начального адреса структуры, представление стандартных типов данных и т.д. По существу, XDR реализуется как независимый пакет, который используется не только в RPC, но и других продуктах (например, в NFS).

Распределенные файловые системы

Основная идея распределенной файловой системы состоит в том, чтобы обеспечить совместный доступ к файлам локальной файловой системы для процессов, которые, вообще говоря, выполняются на других компьютерах. Эта идея может быть реализована многими разными способами, однако в среде ОС UNIX все известные подходы основываются на монтировании удаленной файловой системы к одному из каталогов локальной файловой системы. После выполнения этой процедуры файлы, хранимые в удаленной файловой системе, доступны процессам локального компьютера точно таким же образом, как если бы они хранились на локальном дисковом устройстве.

На рисунке 2.5 приведен пример, в котором два подкаталога удаленной файловой системы-сервера (share и X11) монтируются к двум (пустым) каталогам файловой системы-клиента.

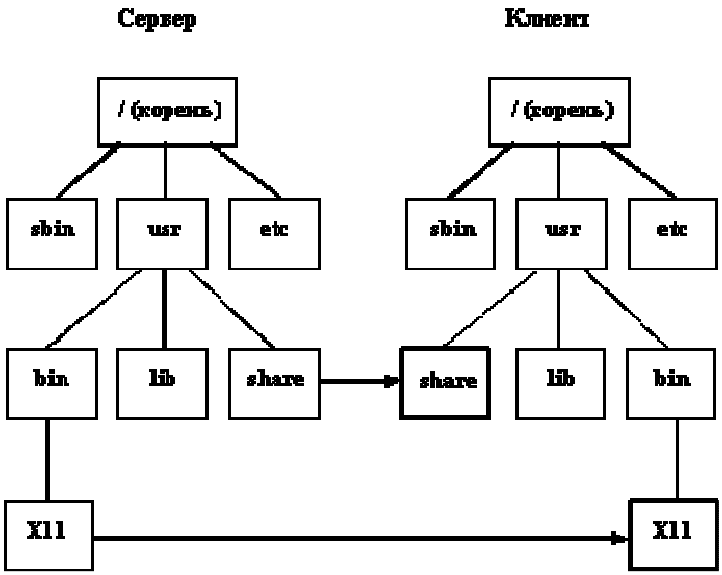

Рис. 2.5. Схема монтирования удаленной файловой системы

В принципе, такая схема обладает и достоинствами, и недостатками. К достоинствам, конечно, относится то, что при работе в сети можно экономить дисковое пространство, поддерживая совместно используемые информационные ресурсы только в одном экземпляре. Но, с другой стороны, пользователи удаленной файловой системы неизбежно будут работать с удаленными файлами существенно более медленно, чем с локальными. Кроме того, реальная возможность доступа к удаленному файлу критически зависит от работоспособности сервера и сети. Заметим, что распространенные в мире UNIX сетевые файловые системы NFS (Network File System - сетевая файловая система) и RFS (Remote File Sharing - совместное использование удаленных файлов) являются достаточно тщательно спроектированными и разработанными продуктами, во многом сглаживающими отмеченные недостатки.

Сетевая файловая система (NFS)

Система NFS была разработана компанией Sun Microsystems как часть ее сетевого продукта ONC (Open Network Computing - открытая сетевая вычислительная обработка). В настоящее время NFS является официальным компонентом UNIX System V Release 4.

NFS разрабатывалась как система, пригодная к использованию не только на разных аппаратных, но и на разных операционных платформах. В настоящее время продукт NFS в соответствии со спецификациями и на основе программного кода Sun Microsystems выпускает более 200 производителей. Отметим, в частности, наличие популярного в России продукта PC-NFS, обеспечивающего клиентскую часть системы в среде MS-DOS. Кроме того, заметим, что имеются и свободно доступные (public domain), и коммерческие варианты NFS.

Первоначально NFS разрабатывалась в среде UNIX BSD 4.2, и для реализации системы потребовалось существенно переделать код системных вызовов файловой системы. При внедрении NFS в среду System V понадобилась значительная переделка ядра ОС. Отмечается, что большая часть изменений в ядре System V Release 4 была связана именно с NFS.

В архитектурном отношении в NFS выделяются три основные части: протокол, серверная часть и клиентская часть. Протокол NFS опирается на примитивы RPC, которые, в свою очередь, построены над протоколом XDR (см. п. 2.7.4). Клиентская часть NFS взаимодействует с серверной частью системы на основе механизма RPC.

Основным достоинством NFS является возможность использования в среде разных операционных систем. Возможным недостатком является то, что независимость от транспортных средств ограничена уровнем такой независимости, присущей RPC. В настоящее время де-факто это означает, что NFS можно использовать только в TCP/IP-ориентированных сетях. (Это еще вопрос - плохо ли это, поскольку стимулирует использование единообразных сетевых механизмов.)

Совместное использование удаленных файлов (RFS)

Сетевая файловая система RFS была реализована компанией AT&T в рамках ее продукта UNIX System V Release 3. Функционально она выглядит подобно NFS, т.е. обеспечивает прозрачный доступ к удаленным файлам. Однако реализация системы абсолютно отлична. Основным недостатком RFS является то, что система реализуема только на компьютерах, работающих под управлением ОС UNIX (причем именно UNIX System V с номером выпуска не меньше, чем 3). Но с другой стороны, это решение позволило сохранить для пользователей RFS всю семантику файлов ОС UNIX. В частности (в отличие от NFS), в удаленной файловой системе могут находиться не только обычные файлы и каталоги, но и специальные файлы и именованные программные каналы. Более того, на удаленные файлы распространяются возможности блокировки файлов и/или диапазонов адресов внутри файлов.

Если NFS опирается на протокол RPC, то в RFS используется родной для AT&T протокол обмена сообщениями на основе потоков (другими словами, реализация RFS основана на использовании интерфейса TLI). (Кстати, в этом имеется большой смысл, поскольку механизм RPC во многих случаях является слишком ограничительным.)

Другим преимуществом RPC (тоже связанным с использованием TLI) является независимость системы от используемого транспортного механизма (если, конечно, этот механизм поддерживает спецификации семиуровневой модели ISO/OSI). Поэтому эту систему можно использовать в среде операционных систем, основывающихся на различных сетевых протоколах (ISO/OSI уважают практически все).

В этой вводной части курса мы (возможно, слишком поверхностно) рассмотрели наиболее важные особенности ОС UNIX. В следующих частях будут детально обсуждаться более технические и/или частные вопросы.

Основные функции и компоненты ядра ОС UNIX

В этой части курса мы более подробно остановимся на базовых функциях ядра ОС UNIX. Основная цель этой части - ввести слушателя курса (и читателя этого документа) в основные идеи ядра ОС UNIX, т.е. показать, чем руководствовались разработчики системы при выборе базовых проектных решений. При этом мы не стремимся излагать технические детали организации ядра, поскольку (как это обычно бывает при попытках совместного идейно-технического изложения) мы утратили бы явное различие между принципиальными и техническими решениями.

Возможно, выбор тем этой части довольно субъективен. Не исключено, что кто-то другой обратил бы большее внимание на другие вопросы, связанные с функциями ядра операционной системы. Однако подчеркнем, что мы следуем классическому представлению о функциях ядра ОС, введенному еще профессором Дейкстрой. В соответствии с этим представлением, ядро любой ОС прежде всего отвечает за управление основной памятью компьютера и виртуальной памятью выполняемых процессов, за управление процессором и планирование распределения процессорных ресурсов между совместно выполняемыми процессами, за управление внешними устройствами и, наконец, за обеспечение базовых средств синхронизации и взаимодействия процессов. Именно эти вопросы мы рассмотрим в данной части курса применительно к ОС UNIX (иногда к UNIX вообще, а иногда к UNIX System V).

Управление памятью

Основная (или как ее принято называть в отечественной литературе и документации, оперативная) память всегда была и остается до сих пор наиболее критическим ресурсом компьютеров. Если учесть, что большинство современных компьютеров обеспечивает 32-разрядную адресацию в пользовательских программах, и все большую силу набирает новое поколение 64-разрядных компьютеров, то становится понятным, что практически безнадежно рассчитывать, что когда-нибудь удастся оснастить компьютеры основной памятью такого объема, чтобы ее хватило для выполнения произвольной пользовательской программы, не говоря уже об обеспечении мультипрограммного режима, когда в основной памяти, вообще говоря, могут одновременно содержаться несколько пользовательских программ.

Поэтому всегда первичной функцией всех операционных систем (более точно, операционных систем, обеспечивающих режим мультипрограммирования) было обеспечение разделения основной памяти между конкурирующими пользовательскими процессами. Мы не будем здесь слишком сильно вдаваться в историю этого вопроса. Заметим лишь, что применявшаяся техника распространяется от статического распределения памяти (каждый процесс пользователя должен полностью поместиться в основной памяти, и система принимает к обслуживанию дополнительные пользовательские процессы до тех пор, пока все они одновременно помещаются в основной памяти), с промежуточным решением в виде "простого своппинга" (система по-прежнему располагает каждый процесс в основной памяти целиком, но иногда на основании некоторого критерия целиком сбрасывает образ некоторого процесса из основной памяти во внешнюю память и заменяет его в основной памяти образом некоторого другого процесса), до смешанных стратегий, основанных на использовании "страничной подкачки по требованию" и развитых механизмов своппинга.

Операционная система UNIX начинала свое существование с применения очень простых методов управления памятью (простой своппинг), но в современных вариантах системы для управления памятью применяется весьма изощренная техника.

Виртуальная память

Идея виртуальной памяти далеко не нова. Сейчас многие полагают, что в основе этой идеи лежит необходимость обеспечения (при поддержке операционной системы) видимости практически неограниченной (32- или 64-разрядной) адресуемой пользовательской памяти при наличии основной памяти существенно меньших размеров. Конечно, этот аспект очень важен. Но также важно понимать, что виртуальная память поддерживалась и на компьютерах с 16-разрядной адресацией, в которых объем основной памяти зачастую существенно превышал 64 Кбайта.

Вспомните хотя бы 16-разрядный компьютер PDP-11/70, к которому можно было подключить до 2 Мбайт основной памяти. Другим примером может служить выдающаяся отечественная ЭВМ БЭСМ-6, в которой при 15-разрядной адресации 6-байтовых (48-разрядных) машинных слов объем основной памяти был доведен до 256 Кбайт. Операционные системы этих компьютеров тем не менее поддерживали виртуальную память, основным смыслом которой являлось обеспечение защиты пользовательских программ одной от другой и предоставление операционной системе возможности динамически гибко перераспределять основную память между одновременно поддерживаемыми пользовательскими процессами.

Хотя известны и чисто программные реализации виртуальной памяти, это направление получило наиболее широкое развитие после получения соответствующей аппаратной поддержки. Идея аппаратной части механизма виртуальной памяти состоит в том, что адрес памяти, вырабатываемый командой, интерпретируется аппаратурой не как реальный адрес некоторого элемента основной памяти, а как некоторая структура, разные поля которой обрабатываются разным образом.

В наиболее простом и наиболее часто используемом случае страничной виртуальной памяти каждая виртуальная память (виртуальная память каждого процесса) и физическая основная память представляются состоящими из наборов блоков или страниц одинакового размера. Для удобства реализации размер страницы всегда выбирается равным числу, являющемуся степенью 2. Тогда, если общая длина виртуального адреса есть N (в последние годы это тоже всегда некоторая степень 2 - 16, 32, 64), а размер страницы есть 2^M), то виртуальный адрес рассматривается как структура, состоящая из двух полей: первое поле занимает (N-M+1) разрядов адреса и задает номер страницы виртуальной памяти, второе поле занимает (M-1) разрядов и задает смещение внутри страницы до адресуемого элемента памяти (в большинстве случаев - байта). Аппаратная интерпретация виртуального адреса показана на рисунке 3.1.

Рисунок иллюстрирует механизм на концептуальном уровне, не вдаваясь в детали по поводу того, что из себя представляет и где хранится таблица страниц. Мы не будем рассматривать возможные варианты, а лишь заметим, что в большинстве современных компьютеров со страничной организацией виртуальной памяти все таблицы страниц хранятся в основной памяти, а быстрота доступа к элементам таблицы текущей виртуальной памяти достигается за счет наличия сверхбыстродействующей буферной памяти (кэша).

Для полноты изложения, но не вдаваясь в детали, заметим, что существуют две другие схемы организации виртуальной памяти: сегментная и сегментно-страничная. При сегментной организации виртуальный адрес по-прежнему состоит из двух полей - номера сегмента и смещения внутри сегмента. Отличие от страничной организации состоит в том, что сегменты виртуальной памяти могут быть разного размера. В элементе таблицы сегментов помимо физического адреса начала сегмента (если виртуальный сегмент содержится в основной памяти) содержится длина сегмента. Если размер смещения в виртуальном адресе выходит за пределы размера сегмента, возникает прерывание. Понятно, что компьютер с сегментной организацией виртуальной памяти можно использовать как компьютер со страничной организацией, если использовать сегменты одного размера.

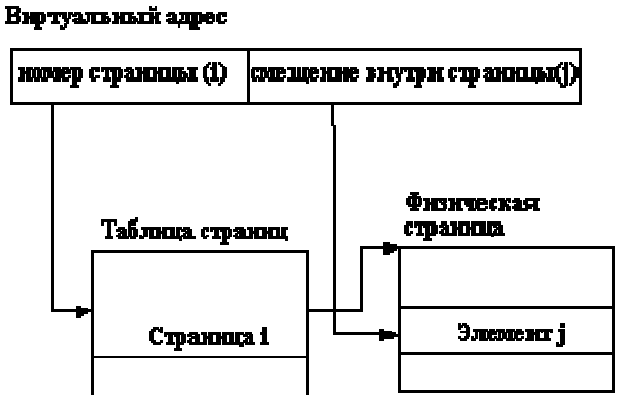

Рис. 3.1. Схема страничной организации виртуальной памяти

При сегментно-страничной организации виртуальной памяти происходит двухуровневая трансляция виртуального адреса в физический. В этом случае виртуальный адрес состоит из трех полей: номера сегмента виртуальной 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 (рисунок 3.2).

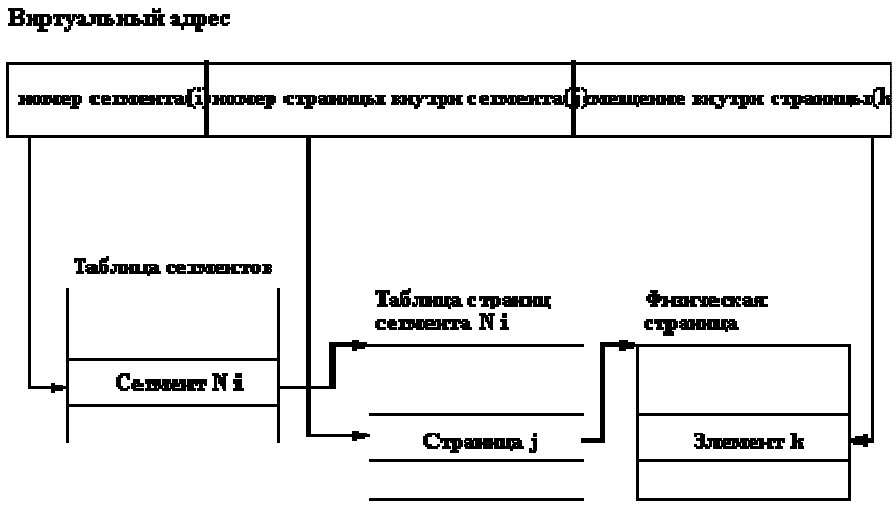

Рис. 3.2. Схема сегментно-страничной организации виртуальной памяти

Сегментно-страничная организация виртуальной памяти позволяла совместно использовать одни и те же сегменты данных и программного кода в виртуальной памяти разных задач (для каждой виртуальной памяти существовала отдельная таблица сегментов, но для совместно используемых сегментов поддерживались общие таблицы страниц).

В дальнейшем рассмотрении мы ограничимся проблемами управления страничной виртуальной памяти. С небольшими коррективами все обсуждаемые ниже методы и алгоритмы относятся и к сегментной, и сегментно-страничной организациям.

Как же достигается возможность наличия виртуальной памяти с размером, существенно превышающим размер оперативной памяти? В элементе таблицы страниц может быть установлен специальный флаг (означающий отсутствие страницы), наличие которого заставляет аппаратуру вместо нормального отображения виртуального адреса в физический прервать выполнение команды и передать управление соответствующему компоненту операционной системы. Английский термин "demand paging" (листание по требованию) достаточно точно характеризует функции, выполняемые этим компонентом. Когда программа обращается к виртуальной странице, отсутствующей в основной памяти, т.е. "требует" доступа к данным или программному коду, операционная система удовлетворяет это требование путем выделения страницы основной памяти, перемещения в нее копии страницы, находящейся во внешней памяти, и соответствующей модификации элемента таблицы страниц. После этого происходит "возврат из прерывания", и команда, по "требованию" которой выполнялись эти действия, продолжает свое выполнение.

Наиболее ответственным действием описанного процесса является выделение страницы основной памяти для удовлетворения требования доступа к отсутствующей в основной памяти виртуальной странице. Напомним, что мы рассматриваем ситуацию, когда размер каждой виртуальной памяти может существенно превосходить размер основной памяти. Это означает, что при выделении страницы основной памяти с большой вероятностью не удастся найти свободную (не приписанную к какой-либо виртуальной памяти) страницу. В этом случае операционная система должна в соответствии с заложенными в нее критериями (совокупность этих критериев принято называть "политикой замещения", а основанный на них алгоритм замещения - "алгоритмом подкачки") найти некоторую занятую страницу основной памяти, переместить в случае надобности ее содержимое во внешнюю память, должным образом модифицировать соответствующий элемент соответствующей таблицы страниц и после этого продолжить процесс удовлетворения доступа к странице.

Существует большое количество разнообразных алгоритмов подкачки. Объем этого курса не позволяет рассмотреть их подробно. Соответствующий материал можно найти в изданных на русском языке книгах по операционным системам Цикритзиса и Бернстайна, Дейтела и Краковяка. Однако, чтобы вернуться к описанию конкретных методов управления виртуальной памятью, применяемых в ОС UNIX, мы все же приведем некоторую краткую классификацию алгоритмов подкачки.

Во-первых, алгоритмы подкачки делятся на глобальные и локальные. При использовании глобальных алгоритмов операционная система при потребности замещения ищет страницу основной памяти среди всех страниц, независимо от их принадлежности к какой-либо виртуальной памяти. Локальные алгоритмы предполагают, что если возникает требование доступа к отсутствующей в основной памяти странице виртуальной памяти ВП1, то страница для замещения будет искаться только среди страниц основной памяти, приписанных к той же виртуальной памяти ВП1.

Наиболее распространенными традиционными алгоритмами (как в глобальном, так в локальном вариантах) являются алгоритмы FIFO (First In First Out) и LRU (Least Recently Used). При использовании алгоритма FIFO для замещения выбирается страница, которая дольше всего остается приписанной к виртуальной памяти. Алгоритм LRU предполагает, что замещать следует ту страницу, к которой дольше всего не происходили обращения. Хотя интуитивно кажется, что критерий алгоритма LRU является более правильным, известны ситуации, в которых алгоритм FIFO работает лучше (и, кроме того, он гораздо более дешево реализуется).

Заметим еще, что при использовании глобальных алгоритмов, вне зависимости от конкретного применяемого алгоритма, возможны и теоретически неизбежны критические ситуации, которые называются по-английски thrashing (несмотря на множество попыток, хорошего русского эквивалента так и не удалось придумать). Рассмотрим простой пример. Пусть на компьютере в мультипрограммном режиме выполняются два процесса - П1 в виртуальной памяти ВП1 и П2 в виртуальной памяти ВП2, причем суммарный размер ВП1 и ВП2 больше размеров основной памяти. Предположим, что в момент времени t1 в процессе П1 возникает требование виртуальной страницы ВС1. Операционная система обрабатывает соответствующее прерывание и выбирает для замещения страницу основной памяти С2, приписанную к виртуальной странице ВС2 виртуальной памяти ВП2 (т.е. в элементе таблицы страниц, соответствующем ВС2, проставляется флаг отсутствия страницы). Для полной обработки требования доступа к ВС1 в общем случае потребуется два обмена с внешней памятью (первый, чтобы записать текущее содержимое С2, второй - чтобы прочитать копию ВС1). Поскольку операционная система поддерживает мультипрограммный режим работы, то во время выполнения обменов доступ к процессору получит процесс П2, и он, вполне вероятно, может потребовать доступа к своей виртуальной странице ВС2 (которую у него только что отняли). Опять будет обрабатываться прерывание, и ОС может заменить некоторую страницу основной памяти С3, которая приписана к виртуальной странице ВС3 в ВП1. Когда закончатся обмены, связанные с обработкой требования доступа к ВС1, возобновится процесс П1, и он, вполне вероятно, потребует доступа к своей виртуальной странице ВС3 (которую у него только что отобрали). И так далее. Общий эффект состоит в том, что непрерывно работает операционная система, выполняя бесчисленные и бессмысленные обмены с внешней памятью, а пользовательские процессы П1 и П2 практически не продвигаются.

Понятно, что при использовании локальных алгоритмов ситуация thrashing, затрагивающая несколько процессов, невозможна. Однако в принципе возможна аналогичная ситуация внутри одной виртуальной памяти: ОС может каждый раз замещать ту страницу, к которой процесс обратится в следующий момент времени.

Единственным алгоритмом, теоретически гарантирующим отсутствие thrashing, является так называемый "оптимальный алгоритм Биледи" (по имени придумавшего его венгерского математика). Алгоритм заключается в том, что для замещения следует выбирать страницу, к которой в будущем наиболее долго не будет обращений. Понятно, что в динамической среде операционной системы точное знание будущего невозможно, и в этом контексте алгоритм Биледи представляет только теоретический интерес (хотя он с успехом применяется практически, например, в компиляторах для планирования использования регистров).

В 1968 году американский исследователь Питер Деннинг сформулировал принцип локальности ссылок (называемый принципом Деннинга) и выдвинул идею алгоритма подкачки, основанного на понятии рабочего набора. В некотором смысле предложенный им подход является практически реализуемой аппроксимацией оптимального алгоритма

Биледи. Принцип локальности ссылок (недоказуемый, но подтверждаемый на практике) состоит в том, что если в период времени (T-t, T) программа обращалась к страницам (C1, C2, ..., Cn), 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 хорошо его прогнозировать исходя из прошлого. Набор страниц (C1, C2, ..., Cn) называется рабочим набором программы (или, правильнее, соответствующего процесса) в момент времени T. Понятно, что с течением времени рабочий набор процесса может изменяться (как по составу страниц, так и по их числу). Идея алгоритма подкачки Деннинга (иногда называемого алгоритмом рабочих наборов) состоит в том, что операционная система в каждый момент времени должна обеспечивать наличие в основной памяти текущих рабочих наборов всех процессов, которым разрешена конкуренция за доступ к процессору. Мы не будем вдаваться в технические детали алгоритма, а лишь заметим следующее. Во-первых, полная реализация алгоритма Деннинга практически гарантирует отсутствие thrashing. Во-вторых, алгоритм реализуем (известна, по меньшей мере, одна его полная реализация, которая однако потребовала специальной аппаратной поддержки). В-третьих, полная реализация алгоритма Деннинга вызывает очень большие накладные расходы.

Поэтому на практике применяются облегченные варианты алгоритмов подкачки, основанных на идее рабочего набора. Один из таких вариантов применяется и в ОС UNIX (насколько нам известно, во всех версиях системы, относящихся к ветви System V). Мы кратко опишем этот вариант в п. 3.1.3.

Аппаратно-независимый уровень управления памятью

Материал, приведенный в данном разделе, хотя и не отражает в полном объеме все проблемы и решения, связанные с управлением виртуальной памятью, достаточен для того, чтобы осознать важность и сложность соответствующих компонентов операционной системы. В любой операционной системе управление виртуальной памятью занимает центральное место. Когда-то Игорь Силин (основной разработчик известной операционной системы Дубна для БЭСМ-6) выдвинул тезис, известный в народе как "Тезис Силина": "Расходы, затраченные на управление виртуальной памятью, окупаются". Я думаю, что любой специалист в области операционных систем согласится с истинностью этого тезиса.

Понятно, что и разработчики ОС UNIX уделяли большое внимание поискам простых и эффективных механизмов управления виртуальной памятью (в области операционных систем абсолютно истинным является утверждение, что любое хорошее решение обязано быть простым). Но основной проблемой было то, что UNIX должен был быть мобильной операционной системой, легко переносимой на разные аппаратные платформы. Хотя на концептуальном уровне все аппаратные механизмы поддержки виртуальной памяти практически эквивалентны, реальные реализации часто весьма различаются. 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 ОС UNIX является грамотное и эффективное разделение средств управления виртуальной памятью на аппаратно-независимую и аппаратно-зависимую части. Коротко рассмотрим, что и каким образом удалось включить в аппаратно-независимую часть подсистемы управления виртуальной памятью ОС UNIX (ниже мы умышленно опускаем технические детали и упрощаем некоторые аспекты).

Основная идея состоит в том, что ОС UNIX опирается на некоторое собственное представление организации виртуальной памяти, которое используется в аппаратно-независимой части подсистемы управления виртуальной памятью и связывается с конкретной аппаратной реализацией с помощью аппаратно-зависимой части. В чем же состоит это абстрактное представление виртуальной памяти?

Во-первых, виртуальная память каждого процесса представляется в виде набора сегментов (рисунок 3.3).

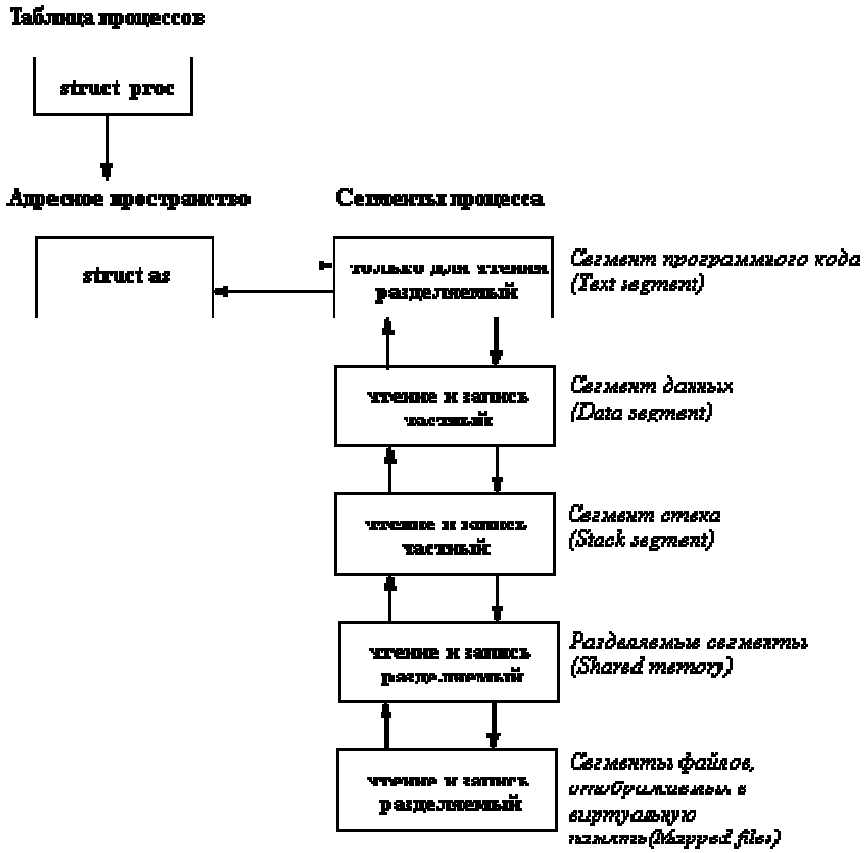

Рис. 3.3. Сегментная структура виртуального адресного пространства

Как видно из рисунка, виртуальная память процесса ОС UNIX разбивается на сегменты пяти разных типов. Три типа сегментов обязательны для каждой виртуальной памяти, и сегменты этих типов присутствуют в виртуальной памяти в одном экземпляре для каждого типа. Сегмент программного кода содержит только команды. Реально в него помещается соответствующий сегмент выполняемого файла, который указывался в качестве параметра системного вызова `exec` для данного процесса. Сегмент программного кода не может модифицироваться в ходе выполнения процесса и потому возможно использование одного экземпляра кода для разных процессов.

Сегмент данных содержит инициализированные и неинициализированные статические переменные программы, выполняемой в данном процессе (на этом уровне изложения под статическими переменными лучше понимать области виртуальной памяти, адреса которых фиксируются в программе при ее загрузке и действуют на протяжении всего ее выполнения). Понятно, что поскольку речь идет о переменных, содержимое сегмента данных может изменяться в ходе выполнения процесса, следовательно, к сегменту должен обеспечиваться доступ и по чтению, и по записи. С другой стороны, поскольку мы

говорим о собственных переменных данной программы, нельзя разрешить нескольким процессам совместно использовать один и тот же сегмент данных (по причине несогласованного изменения одних и тех же переменных разными процессами ни один из них не мог бы успешно завершиться).

Сегмент стека - это область виртуальной памяти, в которой размещаются автоматические переменные программы, явно или неявно в ней присутствующие. Этот сегмент, очевидно, должен быть динамическим (т.е. доступным и по чтению, и по записи), и он, также очевидно, должен быть частным (приватным) сегментом процесса.

Разделяемый сегмент виртуальной памяти образуется при подключении к ней сегмента разделяемой памяти (см. п. 3.4.1). По определению, такие сегменты предназначены для координированного совместного использования несколькими процессами. Поэтому разделяемый сегмент должен допускать доступ по чтению и по записи и может разделяться несколькими процессами.

Сегменты файлов, отображаемых в виртуальную память (см. п. 2.4.5), представляют собой разновидность разделяемых сегментов. Разница состоит в том, что если при необходимости освободить оперативную память страницы разделяемых сегментов копируются ("откачиваются") в специальную системную область подкачки (swapping space) на диске, то страницы сегментов файлов, отображаемых в виртуальную память, в случае необходимости откачиваются прямо на свое место в области внешней памяти, занимаемой файлом. Такие сегменты также допускают доступ и по чтению, и по записи и являются потенциально совместно используемыми.

На аппаратно-независимом уровне сегментная организация виртуальной памяти каждого процесса описывается структурой `as`, которая содержит указатель на список описателей сегментов, общий текущий размер виртуальной памяти (т.е. суммарный размер всех существующих сегментов), текущий размер физической памяти, которую процесс занимает в данный момент времени, и наконец, указатель на некоторую аппаратно-зависимую структуру, данные которой используются при отображении виртуальных адресов в физические. Описатель каждого сегмента (несколько огрубляя) содержит индивидуальные характеристики сегмента, в том числе, виртуальный адрес начала сегмента (каждый сегмент занимает некоторую непрерывную область виртуальной памяти), размер сегмента в байтах, список операций, которые можно выполнять над данным сегментом, статус сегмента (например, в каком режиме к нему возможен доступ, допускается ли совместное использование и т.д.), указатель на таблицу описателей страниц сегмента и т.д. Кроме того, описатель каждого сегмента содержит прямые и обратные ссылки по списку описателей сегментов данной виртуальной памяти и ссылку на общий описатель виртуальной памяти `as`.

На уровне страниц поддерживается два вида описательных структур. Для каждой страницы физической оперативной памяти существует описатель, входящий в один из трех списков. Первый список включает описатели страниц, не допускающих модификации или отображаемых в область внешней памяти какого-либо файла (например, страницы сегментов программного кода или страницы сегмента файла, отображаемого в виртуальную память). Для таких страниц не требуется пространство в области подкачки системы; они либо вовсе не требуют откачки (перемещения копии во внешнюю память), либо откачка производится в другое место. Второй список - это список описателей свободных страниц, т.е. таких страниц, которые не подключены ни к одной виртуальной памяти. Такие страницы свободны для использования и могут быть подключены к любой виртуальной памяти. Наконец, третий список страниц включает

описатели так называемых анонимных страниц, т.е. таких страниц, которые могут изменяться, но для которых нет "родного" места во внешней памяти.

В любом описателе физической страницы сохраняются копии признаков обращения и модификации страницы, вырабатываемых конкретной используемой аппаратурой.

Для каждого сегмента поддерживается таблица отображения, связывающая адреса входящих в него виртуальных страниц с описателями соответствующих им физических страниц из первого или третьего списков описателей физических страниц для виртуальных страниц, присутствующих в основной памяти, или с адресами копий страниц во внешней памяти для виртуальных страниц, отсутствующих в основной памяти. (Правильнее сказать, что поддерживается отдельная таблица отображения для каждого частного сегмента и одна общая таблица отображения для каждого разделяемого сегмента.)

Введение подобной обобщенной модели организации виртуальной памяти и тщательное продумывание связи аппаратно-независимой и аппаратно-зависимой частей подсистемы управления виртуальной памятью позволило добиться того, что обращения к памяти, не требующие вмешательства операционной системы, производятся, как и полагается, напрямую с использованием конкретных аппаратных средств. Вместе с тем, все наиболее ответственные действия операционной системы, связанные с управлением виртуальной памятью, выполняются в аппаратно-независимой части с необходимыми взаимодействиями с аппаратно-зависимой частью.

Конечно, в результате сложность переноса той части ОС UNIX, которая относится к управлению виртуальной памятью, определяется сложностью написания аппаратно-зависимой части. Чем ближе архитектура аппаратуры, поддерживающей виртуальную память, к абстрактной модели виртуальной памяти ОС UNIX, тем проще перенос. Для справедливости заметим, что в подавляющем большинстве современных компьютеров аппаратура выполняет функции, существенно превышающие потребности модели UNIX, так что создание новой аппаратно-зависимой части подсистемы управления виртуальной памятью ОС UNIX в большинстве случаев не является чрезмерно сложной задачей.

Страничное замещение основной памяти и swapping

Как мы упоминали в конце п. 3.1.1, в ОС UNIX используется некоторый облегченный вариант алгоритма подкачки, основанный на использовании понятия рабочего набора. Основная идея заключается в оценке рабочего набора процесса на основе использования аппаратно (а в некоторых реализациях - программно) устанавливаемых признаков обращения к страницам основной памяти. (Заметим, что в этом подразделе при описании алгоритма мы не различаем функции аппаратно-независимого и аппаратно-зависимого компонентов подсистемы управления виртуальной памятью.)

Периодически для каждого процесса производятся следующие действия. Просматриваются таблицы отображения всех сегментов виртуальной памяти этого процесса. Если элемент таблицы отображения содержит ссылку на описатель физической страницы, то анализируется признак обращения. Если признак установлен, то страница считается входящей в рабочий набор данного процесса, и сбрасывается в нуль счетчик старения данной страницы. Если признак не установлен, то к счетчику старения добавляется единица, а страница приобретает статус кандидата на выход из рабочего набора процесса. Если при этом значение счетчика достигает некоторого (различающегося в разных реализациях) критического значения, страница считается вышедшей из рабочего

набора процесса, и ее описатель заносится в список страниц, которые можно откачать (если это требуется) во внешнюю память. По ходу просмотра элементов таблиц отображения в каждом из них признак обращения гасится.

Откачку страниц, не входящих в рабочие наборы процессов, производит специальный системный процесс-stealer. Он начинает работать, когда количество страниц в списке свободных страниц достигает установленного нижнего порога. Функцией этого процесса является анализ необходимости откачки страницы (на основе признака изменения) и запись копии страницы (если это требуется) в соответствующую область внешней памяти (т.е. либо в системную область подкачки - swapping space для анонимных страниц, либо в некоторый блок файловой системы для страницы, входящей в сегмент отображаемого файла).

Очевидно, рабочий набор любого процесса может изменяться во время его выполнения. Другими словами, возможна ситуация, когда процесс обращается к виртуальной странице, отсутствующей в основной памяти. В этом случае, как обычно, возникает аппаратное прерывание, в результате которого начинает работать операционная система. Дальнейший ход событий зависит от обстоятельств. Если список описателей свободных страниц не пуст, то из него выбирается некоторый описатель, и соответствующая страница подключается к виртуальной памяти процесса (конечно, после считывания из внешней памяти содержимого копии этой страницы, если это требуется).

Но если возникает требование страницы в условиях, когда список описателей свободных страниц пуст, то начинает работать механизм своппинга. Основной повод для применения другого механизма состоит в том, что простое отнятие страницы у любого процесса (включая тот, который затребовал бы страницу) потенциально вело бы к ситуации thrashing, поскольку разрушало бы рабочий набор некоторого процесса). Любой процесс, затребовавший страницу не из своего текущего рабочего набора, становится кандидатом на своппинг. Ему больше не предоставляются ресурсы процессора, и описатель процесса ставится в очередь к системному процессу-swapper. Конечно, в этой очереди может находиться несколько процессов. Процесс-swapper по очереди осуществляет полный своппинг этих процессов (т.е. откачку всех страниц их виртуальной памяти, которые присутствуют в основной памяти), помещая соответствующие описатели физических страниц в список свободных страниц, до тех пор, пока количество страниц в этом списке не достигнет установленного в системе верхнего предела. После завершения полного своппинга каждого процесса одному из процессов из очереди к процессу-swapper дается возможность попытаться продолжить свое выполнение (в расчете на то, что свободной памяти уже может быть достаточно).

Заметим, что мы описали наиболее сложный алгоритм, когда бы то ни было использовавшийся в ОС UNIX. В последней "фактически стандартной" версии ОС UNIX (System V Release 4) используется более упрощенный алгоритм. Это глобальный алгоритм, в котором вероятность thrashing погашается за счет своппинга. Используемый алгоритм называется NRU (Not Recently Used) или clock. Смысл алгоритма состоит в том, что процесс-stealer периодически очищает признаки обращения всех страниц основной памяти, входящих в виртуальную память процессов (отсюда название "clock"). Если возникает потребность в откачке (т.е. достигнут нижний предел размера списка описателей свободных страниц), то stealer выбирает в качестве кандидатов на откачку прежде всего те страницы, к которым не было обращений по записи после последней "очистки" и у которых нет признака модификации (т.е. те, которые можно дешевле освободить). Во вторую очередь выбираются страницы, которые действительно нужно откачивать. Параллельно с этим работает описанный выше алгоритм своппинга, т.е. если

возникает требование страницы, а свободных страниц нет, то соответствующий процесс становится кандидатом на своппинг.

В заключение затронем еще одну важную тему, непосредственно связанную с управлением виртуальной памятью - копирование страниц при попытке записи (copy on write). Как мы отмечали в п. 2.1.7, при выполнении системного вызова `fork()` ОС UNIX образует процесс-потомок, являющийся полной копией своего предка. Тем не менее, у потомка своя собственная виртуальная память, и те сегменты, которые должны быть его частными сегментами, в принципе должны были бы полностью скопироваться. Однако, несмотря на то, что частные сегменты допускают доступ и по чтению, и по записи, ОС не знает, будет ли предок или потомок реально производить запись в каждую страницу таких сегментов. Поэтому было бы неразумно производить полное копирование частных сегментов во время выполнения системного вызова `fork()`.

Поэтому в таких случаях используется техника копирования страниц при попытке записи. Несмотря на то, что в сегмент запись разрешена, для каждой его страницы устанавливается блокировка записи. Тем самым, во время попытки выполнения записи возникает прерывание, и ОС на основе анализа статуса соответствующего сегмента принимает решение о выделении новой страницы, копировании на нее содержимого оригинальной страницы и о включении этой новой страницы на место старой в виртуальную память либо процесса-предка, либо процесса-потомка (в зависимости от того, кто из них пытался писать).

На этом мы заканчиваем краткое описание механизма управления виртуальной памятью в ОС UNIX. Еще раз подчеркнем, что мы опустили множество важных технических деталей, стремясь продемонстрировать наиболее важные принципиальные решения.

Управление процессами и нитями

В операционной системе UNIX традиционно поддерживается классическая схема мультипрограммирования. Система поддерживает возможность параллельного (или квази-параллельного в случае наличия только одного аппаратного процессора) выполнения нескольких пользовательских программ. Каждому такому выполнению соответствует процесс операционной системы. Каждый процесс выполняется в собственной виртуальной памяти, и, тем самым, процессы защищены один от другого, т.е. один процесс не в состоянии неконтроллируемым образом прочитать что-либо из памяти другого процесса или записать в нее. Однако контролируемые взаимодействия процессов допускаются системой, в том числе за счет возможности разделения одного сегмента памяти между виртуальной памятью нескольких процессов.

Конечно, не менее важно (а на самом деле, существенно более важно) защищать саму операционную систему от возможности ее повреждения каким бы то ни было пользовательским процессом. В ОС UNIX это достигается за счет того, что ядро системы работает в собственном "ядерном" виртуальном пространстве, к которому не может иметь доступа ни один пользовательский процесс.

Ядро системы предоставляет возможности (набор системных вызовов) для порождения новых процессов, отслеживания окончания порожденных процессов и т.д. С другой стороны, в ОС UNIX ядро системы - это полностью пассивный набор программ и данных. Любая программа ядра может начать работать только по инициативе некоторого пользовательского процесса (при выполнении системного вызова), либо по причине внутреннего или внешнего прерывания (примером внутреннего прерывания может быть

прерывание из-за отсутствия в основной памяти требуемой страницы виртуальной памяти пользовательского процесса; примером внешнего прерывания является любое прерывание процессора по инициативе внешнего устройства). В любом случае считается, что выполняется ядерная часть обратившегося или прерванного процесса, т.е. ядро всегда работает в контексте некоторого процесса.

В последние годы в связи с широким распространением так называемых симметричных мультипроцессорных архитектур компьютеров (Symmetric Multiprocessor Architectures - SMP) в ОС UNIX был внедрен механизм легковесных процессов (light-weight processes), или нитей, или потоков управления (threads). Говоря по-простому, нить - это процесс, выполняющийся в виртуальной памяти, используемой совместно с другими нитями того же "тяжеловесного" (т.е. обладающего отдельной виртуальной памятью) процесса. В принципе, легковесные процессы использовались в операционных системах много лет назад. Уже тогда стало ясно, что программирование с неконтролируемым использованием общей памяти приносит больше хлопот и неприятностей, чем пользы, по причине необходимости использования явных примитивов синхронизации.

Однако, до настоящего времени в практику программистов так и не были внедрены более безопасные методы параллельного программирования, а реальные возможности мультипроцессорных архитектур для обеспечения распараллеливания нужно было как-то использовать. Поэтому опять в обиход вошли легковесные процессы, которые теперь получили название threads (нити). Наиболее важно (с нашей точки зрения) то, что для внедрения механизма нитей потребовалась существенная переделка ядра. Разные производители аппаратуры и программного обеспечения стремились как можно быстрее выставить на рынок продукт, пригодный для эффективного использования на SMP-платформах. Поэтому версии ОС UNIX опять несколько разошлись.

Все эти вопросы мы обсудим более подробно в данном разделе.

Пользовательская и ядерная составляющие процессов

Каждому процессу соответствует контекст, в котором он выполняется. Этот контекст включает содержимое пользовательского адресного пространства - пользовательский контекст (т.е. содержимое сегментов программного кода, данных, стека, разделяемых сегментов и сегментов файлов, отображаемых в виртуальную память), содержимое аппаратных регистров - регистровый контекст (регистр счетчика команд, регистр состояния процессора, регистр указателя стека и регистры общего назначения), а также структуры данных ядра (контекст системного уровня), связанные с этим процессом. Контекст процесса системного уровня в ОС UNIX состоит из "статической" и "динамических" частей. Для каждого процесса имеется одна статическая часть контекста системного уровня и переменное число динамических частей.

Статическая часть контекста процесса системного уровня включает следующее:

1. Описатель процесса, т.е. элемент таблицы описателей существующих в системе процессов. Описатель процесса включает, в частности, следующую информацию:
    - состояние процесса;
    - физический адрес в основной или внешней памяти u-области процесса;
    - идентификаторы пользователя, от имени которого запущен процесс;
    - идентификатор процесса;
    - прочую информацию, связанную с управлением процессом.

2. U-область (u-area) - индивидуальная для каждого процесса область пространства ядра, обладающая тем свойством, что хотя u-область каждого процесса располагается в отдельном месте физической памяти, u-области всех процессов имеют один и тот же виртуальный адрес в адресном пространстве ядра. Именно это означает, что какая бы программа ядра не выполнялась, она всегда выполняется как ядерная часть некоторого пользовательского процесса, и именно того процесса, u-область которого является "видимой" для ядра в данный момент времени. U-область процесса содержит:
    - указатель на описатель процесса;
    - идентификаторы пользователя;
    - счетчик времени, в течение которого процесс реально выполнялся (т.е. занимал процессор) в режиме пользователя и режиме ядра;
    - параметры системного вызова;
    - результаты системного вызова;
    - таблица дескрипторов открытых файлов;
    - предельные размеры адресного пространства процесса;
    - предельные размеры файла, в который процесс может писать;

и т.д.

Динамическая часть контекста процесса - это один или несколько стеков, которые используются процессом при его выполнении в режиме ядра. Число ядерных стеков процесса соответствует числу уровней прерывания, поддерживаемых конкретной аппаратурой.

Принципы организации многопользовательского режима

Основной проблемой организации многопользовательского (правильнее сказать, мультипрограммного) режима в любой операционной системе является организация планирования "параллельного" выполнения нескольких процессов. Операционная система должна обладать четкими критериями для определения того, какому готовому к выполнению процессу и когда предоставить ресурс процессора.

Исторически ОС UNIX является системой разделения времени, т.е. система должна прежде всего "справедливо" разделять ресурсы процессора(ов) между процессами, относящимися к разным пользователям, причем таким образом, чтобы время реакции каждого действия интерактивного пользователя находилось в допустимых пределах. Однако в последнее время возрастает тенденция к использованию ОС UNIX в приложениях реального времени, что повлияло и на алгоритмы планирования. Ниже мы опишем общую (без технических деталей) схему планирования разделения ресурсов процессора(ов) между процессами в UNIX System V Release 4.

Наиболее распространенным алгоритмом планирования в системах разделения времени является кольцевой режим (round robin). Основной смысл алгоритма состоит в том, что время процессора делится на кванты фиксированного размера, а процессы, готовые к выполнению, выстраиваются в кольцевую очередь. У этой очереди имеются два указателя - начала и конца. Когда процесс, выполняющийся на процессоре, исчерпывает свой квант процессорного времени, он снимается с процессора, ставится в конец очереди, а ресурсы процессора отдаются процессу, находящемуся в начале очереди. Если выполняющийся на процессоре процесс откладывается (например, по причине обмена с некоторым внешнем устройством) до того, как он исчерпает свой квант, то после повторной активизации он становится в конец очереди (не смог доработать - не вина системы). Это прекрасная схема

разделения времени в случае, когда все процессы одновременно помещаются в оперативной памяти.

Однако ОС UNIX всегда была рассчитана на то, чтобы обслуживать больше процессов, чем можно одновременно разместить в основной памяти. Другими словами, часть процессов, потенциально готовых выполняться, размещалась во внешней памяти (куда образ памяти процесса попадал в результате свопинга). Поэтому требовалась несколько более гибкая схема планирования разделения ресурсов процессора(ов). В результате было введено понятие приоритета. В ОС UNIX значение приоритета определяет, во-первых, возможность процесса пребывать в основной памяти и на равных конкурировать за процессор. Во-вторых, от значения приоритета процесса, вообще говоря, зависит размер временного кванта, который предоставляется процессу для работы на процессоре при достижении своей очереди. В-третьих, значение приоритета, влияет на место процесса в общей очереди процессов к ресурсу процессора(ов).

Схема разделения времени между процессами с приоритетами в общем случае выглядит следующим образом. Готовые к выполнению процессы выстраиваются в очередь к процессору в порядке уменьшения своих приоритетов. Если некоторый процесс отработал свой квант процессорного времени, но при этом остался готовым к выполнению, то он становится в очередь к процессору впереди любого процесса с более низким приоритетом, но вслед за любым процессом, обладающим тем же приоритетом. Если некоторый процесс активизируется, то он также ставится в очередь вслед за процессом, обладающим тем же приоритетом. Весь вопрос в том, когда принимать решение о своппинге процесса, и когда возвращать в оперативную память процесс, содержимое памяти которого было ранее перемещено во внешнюю память.

Традиционное решение ОС UNIX состоит в использовании динамически изменяющихся приоритетов. Каждый процесс при своем образовании получает некоторый устанавливаемый системой статический приоритет, который в дальнейшем может быть изменен с помощью системного вызова `nice` (см. п. 3.1.3). Этот статический приоритет является основой начального значения динамического приоритета процесса, являющегося реальным критерием планирования. Все процессы с динамическим приоритетом не ниже порогового участвуют в конкуренции за процессор (по схеме, описанной выше). Однако каждый раз, когда процесс успешно отрабатывает свой квант на процессоре, его динамический приоритет уменьшается (величина уменьшения зависит от статического приоритета процесса). Если значение динамического приоритета процесса достигает некоторого нижнего предела, он становится кандидатом на откачку (своппинг) и больше не конкурирует за процессор.

Процесс, образ памяти которого перемещен во внешнюю память, также обладает динамическим приоритетом. Этот приоритет не дает процессу право конкурировать за процессор (да это и невозможно, поскольку образ памяти процесса не находится в основной памяти), но он изменяется, давая в конце концов процессу возможность вновь вернуться в основную память и принять участие в конкуренции за процессор. Правила изменения динамического приоритета для процесса, перемещенного во внешнюю память, в принципе, очень просты. Чем дольше образ процесса находится во внешней памяти, тем более высок его динамический приоритет (конкретное значение динамического приоритета, конечно, зависит от его статического приоритета). Конечно, раньше или позже значение динамического приоритета такого процесса перешагнет через некоторый порог, и тогда система принимает решение о необходимости возврата образа процесса в основную память. После того, как в результате своппинга будет освобождена достаточная по размерам область основной памяти, процесс с приоритетом, достигшим критического

значения, будет перемещен в основную память и будет в соответствии со своим приоритетом конкурировать за процессор.

Как вписываются в эту схему процессы реального времени? Прежде всего, нужно разобраться, что понимается под концепцией "реального времени" в ОС UNIX. Известно, что существуют по крайней мере два понимания термина - " мягкое реальное время (soft realtime)" и " жесткое реальное время (hard realtime)".

Жесткое реальное время означает, что каждое событие (внутреннее или внешнее), происходящее в системе (обращение к ядру системы за некоторой услугой, прерывание от внешнего устройства и т.д.), должно обрабатываться системой за время, не превосходящее верхнего предела времени, отведенного для таких действий. Режим жесткого реального времени требует задания четких временных характеристик процессов, и эти временные характеристики должны определять поведение планировщика распределения ресурсов процессора(ов) и основной памяти.

Режим мягкого реального времени, в отличие от этого, предполагает, что некоторые процессы (процессы реального времени) получают права на получение ресурсов основной памяти и процессора(ов), существенно превосходящие права процессов, не относящихся к категории процессов реального времени. Основная идея состоит в том, чтобы дать возможность процессу реального времени опередить в конкуренции за вычислительные ресурсы любой другой процесс, не относящийся к категории процессов реального времени. Отслеживание проблем конкуренции между различными процессами реального времени относится к функциям администратора системы и выходит за пределы этого курса.

В своих самых последних вариантах ОС UNIX поддерживает концепцию мягкого реального времени. Это делается способом, не выходящим за пределы основополагающего принципа разделения времени. Как мы отмечали выше, имеется некоторый диапазон значений статических приоритетов процессов. Некоторый поддиапазон этого диапазона включает значения статических приоритетов процессов реального времени. Процессы, обладающие динамическими приоритетами, основанными на статических приоритетах процессов реального времени, обладают следующими особенностями:

1. Каждому из таких процессов предоставляется неограниченный сверху квант времени на процессоре. Другими словами, занявший процессор процесс реального времени не будет с него снят до тех пор, пока сам не заявит о невозможности продолжения выполнения (например, задав обмен с внешним устройством).
2. Процесс реального времени не может быть перемещен из основной памяти во внешнюю, если он готов к выполнению, и в оперативной памяти присутствует хотя бы один процесс, не относящийся к категории процессов реального времени (т.е. процессы реального времени перемещаются во внешнюю память последними, причем в порядке убывания своих динамических приоритетов).
3. Любой процесс реального времени, перемещенный во внешнюю память, но готовый к выполнению, переносится обратно в основную память как только в ней образуется свободная область соответствующего размера. (Выбор процесса реального времени для возвращения в основную память производится на основании значений динамических приоритетов.)

Тем самым своеобразным, но логичным образом в современных вариантах ОС UNIX одновременно реализована как возможность разделения времени для интерактивных

процессов, так и возможность мягкого реального времени для процессов, связанных с реальным управлением поведением объектов в реальном времени.

Традиционный механизм управления процессами на уровне пользователя

Как свойственно операционной системе UNIX вообще, имеются две возможности управления процессами - с использованием командного языка (того или другого варианта Shell) и с использованием языка программирования с непосредственным использованием системных вызовов ядра операционной системы. Возможности командных языков мы будем обсуждать в пятой части этого курса, а пока сосредоточимся на базовых возможностях управления процессами на пользовательском уровне, предоставляемых ядром системы.

Прежде всего обрисуем общую схему возможностей пользователя, связанных с управлением процессами. Каждый процесс может образовать полностью идентичный подчиненный процесс с помощью системного вызова `fork()` и дожидаться окончания выполнения своих подчиненных процессов с помощью системного вызова `wait`. Каждый процесс в любой момент времени может полностью изменить содержимое своего образа памяти с помощью одной из разновидностей системного вызова `exec` (сменить образ памяти в соответствии с содержимым указанного файла, хранящего образ процесса (выполняемого файла)). Каждый процесс может установить свою собственную реакцию на "сигналы", производимые операционной системой в соответствие с внешними или внутренними событиями. Наконец, каждый процесс может повлиять на значение своего статического (а тем самым и динамического) приоритета с помощью системного вызова `nice`.

Для создания нового процесса используется системный вызов `fork`. В среде программирования нужно относиться к этому системному вызову как к вызову функции, возвращающей целое значение - идентификатор порожденного процесса, который затем может использоваться для управления (в ограниченном смысле) порожденным процессом. Реально, все процессы системы UNIX, кроме начального, запускаемого при раскрутке системы, образуются при помощи системного вызова `fork`.

Вот что делает ядро системы при выполнении системного вызова `fork`:

1. Выделяет память под описатель нового процесса в таблице описателей процессов.
2. Назначает уникальный идентификатор процесса (PID) для вновь образованного процесса.
3. Образует логическую копию процесса, выполняющего системный вызов `fork`, включая полное копирование содержимого виртуальной памяти процесса-предка во вновь создаваемую виртуальную память, а также копирование составляющих ядерного статического и динамического контекстов процесса-предка.
4. Увеличивает счетчики открытия файлов (процесс-потомок автоматически наследует все открытые файлы своего родителя).
5. Возвращает вновь образованный идентификатор процесса в точку возврата из системного вызова в процессе-предке и возвращает значение 0 в точке возврата в процессе-потомке.

Понятно, что после создания процесса предок и потомок начинают жить своей собственной жизнью, произвольным образом изменяя свой контекст. В частности, и

предок, и потомок могут выполнить какой-либо из вариантов системного вызова `exec` (см. ниже), приводящего к полному изменению контекста процесса.

Чтобы процесс-предок мог синхронизовать свое выполнение с выполнением своих процессов-потомков, существует системный вызов `wait`. Выполнение этого системного вызова приводит к приостановке выполнения процесса до тех пор, пока не завершится выполнение какого-либо из процессов, являющихся его потомками. В качестве прямого параметра системного вызова `wait` указывается адрес памяти (указатель), по которому должна быть возвращена информация, описывающая статус завершения очередного процесса-потомка, а ответным (возвратным) параметром является PID (идентификатор процесса) завершившегося процесса-потомка.

Сигнал - это способ информирования процесса со стороны ядра о происшествии некоторого события. Смысл термина "сигнал" состоит в том, что сколько бы однотипных событий в системе не произошло, по поводу каждой такой группы событий процессу будет подан ровно один сигнал. Т.е. сигнал означает, что определяемое им событие произошло, но не несет информации о том, сколько именно произошло однотипных событий.

Примерами сигналов (не исчерпывающими все возможности) являются следующие:

- окончание процесса-потомка (по причине выполнения системного вызова `exit` или системного вызова `signal` с параметром "death of child (смерть потомка)";
- возникновение исключительной ситуации в поведении процесса (выход за допустимые границы виртуальной памяти, попытка записи в область виртуальной памяти, которая доступна только для чтения и т.д.);
- превышение верхнего предела системных ресурсов;
- оповещение об ошибках в системных вызовах (несуществующий системный вызов, ошибки в параметрах системного вызова, несоответствие системного вызова текущему состоянию процесса и т.д.);
- сигналы, посылаемые другим процессом в пользовательском режиме (см. ниже);
- сигналы, поступающие вследствие нажатия пользователем определенных клавишей на клавиатуре терминала, связанного с процессом (например, Ctrl-C или Ctrl-D);
- сигналы, служащие для трассировки процесса.

Для установки реакции на поступление определенного сигнала используется системный вызов

```
oldfunction = signal(signum, function),
```

где `signum` - это номер сигнала, на поступление которого устанавливается реакция (все возможные сигналы пронумерованы, каждому номеру соответствует собственное символическое имя; соответствующая информация содержится в документации к каждой конкретной системе UNIX), а `function` - адрес указываемой пользователем функции, которая должна быть вызвана системой при поступлении указанного сигнала данному процессу. Возвращаемое значение `oldfunction` - это адрес функции для реагирования на поступление сигнала `signum`, установленный в предыдущем вызове `signal`. Вместо адреса функции во входных параметрах можно задать 1 или 0. Задание единицы приводит к тому, что далее для данного процесса поступление сигнала с номером `signum` будет игнорироваться (это допускается не для всех сигналов). Если в качестве значения параметра `function` указан нуль, то после выполнения системного вызова `signal` первое

же поступление данному процессу сигнала с номером `signum` приведет к завершению процесса (будет проинтерпретировано аналогично выполнению системного вызова `exit`, см. ниже).

Система предоставляет возможность для пользовательских процессов явно генерировать сигналы, направляемые другим процессам. Для этого используется системный вызов

```
kill(pid, signum)
```

(Этот системный вызов называется "kill", потому что наиболее часто применяется для того, чтобы принудительно закончить указанный процесс.) Параметр `signum` задает номер генерируемого сигнала (в системном вызове `kill` можно указывать не все номера сигналов). Параметр `pid` имеет следующие смысл:

- если в качестве значения `pid` указано целое положительное число, то ядро пошлет указанный сигнал процессу, идентификатор которого равен `pid`;
- если значение `pid` равно нулю, то указанный сигнал посылается всем процессам, относящимся к той же группе процессов, что и посылающий процесс (понятие группы процессов аналогично понятию группы пользователей; полный идентификатор процесса состоит из двух частей - идентификатора группы процессов и индивидуального идентификатора процесса; в одну группу автоматически включаются все процессы, имеющие общего предка; идентификатор группы процесса может быть изменен с помощью системного вызова `setpgrp`);
- если значение `pid` равно -1, то ядро посылает указанный сигнал всем процессам, действительный идентификатор пользователя которых равен идентификатору текущего выполнения процесса, посылающего сигнал (см. п. 2.5.1).

Для завершения процесса по его собственной инициативе используется системный вызов

```
exit(status),
```

где `status` - это целое число, возвращаемое процессу-предку для его информирования о причинах завершения процесса-потомка (как описывалось выше, для получения информации о статусе завершившегося процесса-потомка в процессе-предке используется системный вызов `wait`). Системный вызов называется `exit` (т.е. "выход", поскольку считается, что любой пользовательский процесс запускается ядром системы (собственно, так оно и есть), и завершение пользовательского процесса - это окончательный выход из него в ядро.

Системный вызов `exit` может задаваться явно в любой точке процесса, но может быть и неявным. В частности, при программировании на языке Си возврат из функции main приводит к выполнению неявно содержащегося в программе системного вызова `exit` с некоторым предопределенным статусом. Кроме того, как отмечалось выше, можно установить такую реакцию на поступающие в процесс сигналы, когда приход определенного сигнала будет интерпретироваться как неявный вызов `exit`. В этом случае в качестве значения статуса указывается номер сигнала.

Возможности управления процессами, которые мы обсудили до этого момента, позволяют образовать новый процесс, являющийся полной копией процесса-предка (системный вызов `fork`); дожидаться завершения любого образованного таким образом процесса (системный вызов `wait`); порождать сигналы и реагировать на них (системные вызовы `kill` и `signal`); завершать процесс по его собственной инициативе (системный вызов

exit). Однако пока остается непонятным, каким образом можно выполнить в образованном процессе произвольную программу. Понятно, что в принципе этого можно было бы добиться с помощью системного вызова fork, если образ памяти процесса-предка заранее построен так, что содержит все потенциально требуемые программы. Но, конечно, этот способ не является рациональным (хотя бы потому, что заведомо приводит к перерасходу памяти).

Для выполнения произвольной программы в текущем процессе используются системные вызовы семейства exec. Разные варианты exec слегка различаются способом задания параметров. Здесь мы не будем вдаваться в детали (за ними лучше обращаться к документации по конкретной реализации). Рассмотрим некоторый обобщенный случай системного вызова

```
exec(filename, argv, argc, envp)
```

Вот что происходит при выполнении этого системного вызова. Берется файл с именем filename (может быть указано полное или сокращенное имя файла). Этот файл должен быть выполняемым файлом, т.е. представлять собой законченный образ виртуальной памяти. Если это на самом деле так, то ядро ОС UNIX производит реорганизацию виртуальной памяти процесса, обратившегося к системному вызову exec, уничтожая в нем старые сегменты и образуя новые сегменты, в которые загружаются соответствующие разделы файла filename. После этого во вновь образованном пользовательском контексте вызывается функция main, которой, как и полагается, передаются параметры argv и argc, т.е. некоторый набор текстовых строк, заданных в качестве параметра системного вызова exec. Через параметр envp обновленный процесс может обращаться к переменным своего окружения.

Следует заметить, что при выполнении системного вызова exec не образуется новый процесс, а лишь меняется содержимое виртуальной памяти существующего процесса. Другими словами, меняется только пользовательский контекст процесса.

Полезные возможности ОС UNIX для общения родственных или независимо образованных процессов рассматриваются ниже в разделе 3.4.

Понятие нити (threads)

Понятие "легковесного процесса" (light-weight process), или, как принято называть его в современных вариантах ОС UNIX, "thread" (нить, поток управления) давно известно в области операционных систем. Интуитивно понятно, что концепции виртуальной памяти и потока команд, выполняющегося в этой виртуальной памяти, в принципе, ортогональны. Ни из чего не следует, что одной виртуальной памяти должен соответствовать один и только один поток управления. Поэтому, например, в ОС Multics (раздел 1.1) допускалось (и являлось принятой практикой) иметь произвольное количество процессов, выполняемых в общей (разделяемой) виртуальной памяти.

Понятно, что если несколько процессов совместно пользуются некоторыми ресурсами, то при доступе к этим ресурсам они должны синхронизоваться (например, с использованием семафоров, см. п. 3.4.2). Многолетний опыт программирования с использованием явных примитивов синхронизации показал, что этот стиль "параллельного" программирования порождает серьезные проблемы при написании, отладке и сопровождении программ (наиболее трудно обнаруживаемые ошибки в программах обычно связаны с синхронизацией). Это явилось одной из причин того, что в традиционных вариантах ОС

UNIX понятие процесса жестко связывалось с понятием отдельной и недоступной для других процессов виртуальной памяти. Каждый процесс был защищен ядром операционной системы от неконтролируемого вмешательства других процессов. Многие годы авторы ОС UNIX считали это одним из основных достоинств системы (впрочем, это мнение существует и сегодня).

Однако, связывание процесса с виртуальной памятью порождает, по крайней мере, две проблемы. Первая проблема связана с так называемыми системами реального времени. Такие системы, как правило, предназначены для одновременного управления несколькими внешними объектами и наиболее естественно представляются в виде совокупности "параллельно" (или "квази-параллельно") выполняемых потоков команд (т.е. взаимодействующих процессов). Однако, если с каждым процессом связана отдельная виртуальная память, то смена контекста процессора (т.е. его переключение с выполнения одного процесса на выполнение другого процесса) является относительно дорогостоящей операцией. Поэтому традиционный подход ОС UNIX препятствовал использованию системы в приложениях реального времени.

Второй (и может быть более существенной) проблемой явилось появление так называемых симметричных мультипроцессорных компьютерных архитектур (SMP - Symmetric Multiprocessor Architectures). В таких компьютерах физически присутствуют несколько процессоров, которые имеют одинаковые по скорости возможности доступа к совместно используемой основной памяти. Появление подобных машин на мировом рынке, естественно, поставило проблему их эффективного использования. Понятно, что при применении традиционного подхода ОС UNIX к организации процессов от наличия общей памяти не очень много толка (хотя при наличии возможностей разделяемой памяти (см. п. 3.4.1) об этом можно спорить). К моменту появления SMP выяснилось, что технология программирования все еще не может предложить эффективного и безопасного способа реального параллельного программирования. Поэтому пришлось вернуться к явному параллельному программированию с использованием параллельных процессов в общей виртуальной (а тем самым, и основной) памяти с явной синхронизацией.

Что же понимается под "нитью" (thread)? Это независимый поток управления, выполняемый в контексте некоторого процесса. Фактически, понятие контекста процесса, которое мы обсуждали в п. 3.1.1, изменяется следующим образом. Все, что не относится к потоку управления (виртуальная память, дескрипторы открытых файлов и т.д.), остается в общем контексте процесса. Вещи, которые характерны для потока управления (регистровый контекст, стеки разного уровня и т.д.), переходят из контекста процесса в контекст нити. Общая картина показана на рисунке 3.4.

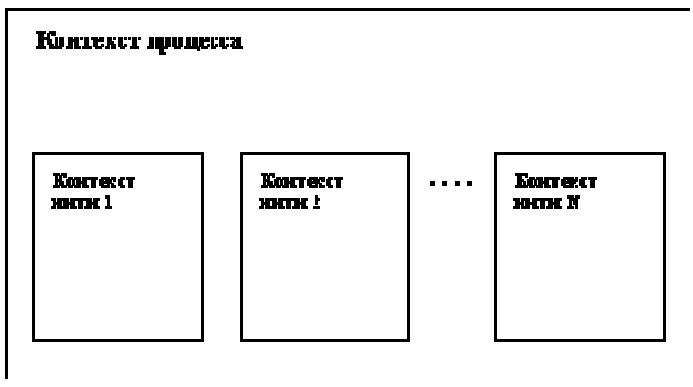

Рис. 3.4. Соотношение контекста процесса и контекстов нитей

Как видно из этого рисунка, все нити процесса выполняются в его контексте, но каждая нить имеет свой собственный контекст. Контекст нити, как и контекст процесса, состоит из пользовательской и ядерной составляющих. Пользовательская составляющая контекста нити включает индивидуальный стек нити. Поскольку нити одного процесса выполняются в общей виртуальной памяти (все нити процесса имеют равные права доступа к любым частям виртуальной памяти процесса), стек (сегмент стека) любой нити процесса в принципе не защищен от произвольного (например, по причине ошибки) доступа со стороны других нитей. Ядерная составляющая контекста нити включает ее регистровый контекст (в частности, содержимое регистра счетчика команд) и динамически создаваемые ядерные стеки.

Приведенное краткое обсуждение понятия нити кажется достаточным для того, чтобы понять, что внедрение в ОС UNIX механизма легковесных процессов требует существенных переделок ядра системы. (Всегда трудно внедрить в программу средства, для поддержки которых она не была изначально приспособлена.)

Подходы к организации нитей и управлению ими в разных вариантах ОС UNIX

Хотя концептуально реализации механизма нитей в разных современных вариантах практически эквивалентны (да и что особенное можно придумать по поводу легковесных процессов?), технически и, к сожалению, в отношении интерфейсов эти реализации различаются. Мы не ставим здесь перед собой цели описать в деталях какую-либо реализацию, однако постараемся в общих чертах охарактеризовать разные подходы.

Начнем с того, что разнообразие механизмов нитей в современных вариантах ОС UNIX само по себе представляет проблему. Сейчас достаточно трудно говорить о возможности мобильного параллельного программирования в среде UNIX-ориентированных операционных систем. Если программист хочет добиться предельной эффективности (а он должен этого хотеть, если для целей его проекта приобретен дорогостоящий мультипроцессор), то он вынужден использовать все уникальные возможности используемой им операционной системы.

Для всех очевидно, что сегодняшняя ситуация далека от идеальной. Однако, по-видимому, ее было невозможно избежать, поскольку поставщики мультипроцессорных симметричных архитектур должны были как можно раньше предоставить своим покупателям возможности эффективного программирования, и времени на согласование решений просто не было (любых поставщиков прежде всего интересует объем продаж, а проблемы будущего оставляются на будущее).

Применяемые в настоящее время подходы зависят от того, насколько внимательно разработчики ОС относились к проблемам реального времени. (Возвращаясь к введению этого раздела, еще раз отметим, что здесь мы имеем в виду "мягкое" реальное время, т.е. программно-аппаратные системы, которые обеспечивают быструю реакцию на внешние события, но время реакции не установлено абсолютно строго.) Типичная система реального времени состоит из общего монитора, который отслеживает общее состояние системы и реагирует на внешние и внутренние события, и совокупности обработчиков событий, которые, желательно параллельно, выполняют основные функции системы.

Понятно, что от возможностей реального распараллеливания функций обработчиков зависят общие временные показатели системы. Если, например, при проектировании системы замечено, что типичной картиной является "одновременное" поступление в

систему N внешних событий, то желательно гарантировать наличие реальных N устройств обработки, на которых могут базироваться обработчики. На этих наблюдениях основан подход компании Sun Microsystems.

В системе Solaris (правильнее говорить SunOS 4.x, поскольку Solaris в терминологии Sun представляет собой не операционную систему, а расширенную операционную среду) принят следующий подход. При запуске любого процесса можно потребовать резервирования одного или нескольких процессоров мультипроцессорной системы. Это означает, что операционная система не предоставит никакому другому процессу возможности выполнения на зарезервированном(ых) процессоре(ах). Независимо от того, готова ли к выполнению хотя бы одна нить такого процесса, зарезервированные процессоры не будут использоваться ни для чего другого.

Далее, при образовании нити можно закрепить ее за одним или несколькими процессорами из числа зарезервированных. В частности, таким образом в принципе можно привязать нить к некоторому фиксированному процессору. В общем случае некоторая совокупность потоков управления привязывается к некоторой совокупности процессоров так, чтобы среднее время реакции системы реального времени удовлетворяло внешним критериям. Очевидно, что это "ассемблерный" стиль программирования (слишком много перекладывается на пользователя), но зато он открывает широкие возможности перед разработчиками систем реального времени (которые, правда, после этого зависят не только от особенностей конкретной операционной системы, но и от конкретной конфигурации данной компьютерной установки). Подход Solaris преследует цели удовлетворить разработчиков систем "мягкого" (а, возможно, и "жесткого") реального времени, и поэтому фактически дает им в руки средства распределения критических вычислительных ресурсов.

В других подходах в большей степени преследуется цель равномерной балансировки загрузки мультипроцессора. В этом случае программисту не предоставляются средства явной привязки процессоров к процессам или нитям. Система допускает явное распараллеливание в пределах общей виртуальной памяти и "обещает", что по мере возможностей все процессоры вычислительной системы будут загружены равномерно. Этот подход обеспечивает наиболее эффективное использование общих вычислительных ресурсов мультипроцессора, но не гарантирует корректность выполнения систем реального времени (если не считать возможности установления специальных приоритетов реального времени, которые упоминались в п. 3.1.2).

Отметим существование еще одной аппаратно-программной проблемы, связанной с нитями (и не только с ними). Проблема связана с тем, что в существующих симметричных мультипроцессорах обычно каждый процессор обладает собственной сверхбыстродействующей буферной памятью (кэшем). Идея кэша, в общих чертах, состоит в том, чтобы обеспечить процессору очень быстрый (без необходимости выхода на шину доступа к общей оперативной памяти) доступ к наиболее актуальным данным. В частности, если программа выполняет запись в память, то это действие не обязательно сразу отображается в соответствующем элементе основной памяти; до поры до времени измененный элемент данных может содержаться только в локальном кэше того процессора, на котором выполняется программа. Конечно, это противоречит идее совместного использования виртуальной памяти нитями одного процесса (а также идее использования памяти, разделяемой между несколькими процессами, см. п. 3.4.1).

Это очень сложная проблема, относящаяся к области проблем "когерентности кэшей". Теоретически имеется много подходов к ее решению (например, аппаратное

распознавание необходимости выталкивания записи из кэша с синхронным объявлением недействительным содержания всех кэшей, включающих тот же элемент данных). Однако на практике такие сложные действия не применяются, и обычным приемом является отмена режима кэширования в том случае, когда на разных процессорах мультипроцессорной системы выполняются нити одного процесса или процессы, использующие разделяемую память.

После введения понятия нити трансформируется само понятие процесса. Теперь лучше (и правильнее) понимать процесс ОС UNIX как некоторый контекст, включающий виртуальную память и другие системные ресурсы (включая открытые файлы), в котором выполняется, по крайней мере, один поток управления (нить), обладающий своим собственным (более простым) контекстом. Теперь ядро знает о существовании этих двух уровней контекстов и способно сравнительно быстро изменять контекст нити (не изменяя общего контекста процесса) и так же, как и ранее, изменять контекст процесса.

Последнее замечание относится к синхронизации выполнения нитей одного процесса (точнее было бы говорить о синхронизации доступа нитей к общим ресурсам процесса - виртуальной памяти, открытым файлам и т.д.). Конечно, можно пользоваться (сравнительно) традиционными средствами синхронизации (например, семафорами, см. п. 3.4.2). Однако оказывается, что система может предоставить для синхронизации нитей одного процесса более дешевые средства (поскольку все нити работают в общем контексте процесса). Обычно эти средства относятся к классу средств взаимного исключения (т.е. к классу семафоро-подобных средств). К сожалению, и в этом отношении к настоящему времени отсутствует какая-либо стандартизация.

Управление вводом/выводом

Мы уже обсуждали проблемы организации ввода/вывода в ОС UNIX в п. 2.6.2. В этом разделе мы хотим рассмотреть этот вопрос немного более подробно, разъяснив некоторые технические детали. При этом нужно отдавать себе отчет, что в любом случае мы остаемся на концептуальном уровне. Если вам требуется написать драйвер некоторого внешнего устройства для некоторого конкретного варианта ОС UNIX, то неизбежно придется внимательно читать документацию. Тем не менее знание общих принципов будет полезно.

Традиционно в ОС UNIX выделяются три типа организации ввода/вывода и, соответственно, три типа драйверов. Блочный ввод/вывод главным образом предназначен для работы с каталогами и обычными файлами файловой системы, которые на базовом уровне имеют блочную структуру. В пп. 2.4.5 и 3.1.2 указывалось, что на пользовательском уровне теперь возможно работать с файлами, прямо отображая их в сегменты виртуальной памяти. Эта возможность рассматривается как верхний уровень блочного ввода/вывода. На нижнем уровне блочный ввод/вывод поддерживается блочными драйверами. Блочный ввод/вывод, кроме того, поддерживается системной буферизацией (см. п. 3.3.1).

Символьный ввод/вывод служит для прямого (без буферизации) выполнения обменов между адресным пространством пользователя и соответствующим устройством. Общей для всех символьных драйверов поддержкой ядра является обеспечение функций пересылки данных между пользовательскими и ядерным адресными пространствами.

Наконец, потоковый ввод/вывод (который мы не будем рассматривать в этом курсе слишком подробно по причине обилия технических деталей) похож на символьный

ввод/вывод, но по причине наличия возможности включения в поток промежуточных обрабатывающих модулей обладает существенно большей гибкостью.

Принципы системной буферизации ввода/вывода

Традиционным способом снижения накладных расходов при выполнении обменов с устройствами внешней памяти, имеющими блочную структуру, является буферизация блочного ввода/вывода. Это означает, что любой блок устройства внешней памяти считывается прежде всего в некоторый буфер области основной памяти, называемой в ОС UNIX системным кэшем, и уже оттуда полностью или частично (в зависимости от вида обмена) копируется в соответствующее пользовательское пространство.

Принципами организации традиционного механизма буферизации является, во-первых, то, что копия содержимого блока удерживается в системном буфере до тех пор, пока не возникнет необходимость ее замещения по причине нехватки буферов (для организации политики замещения используется разновидность алгоритма LRU, см. п. 3.1.1). Во-вторых, при выполнении записи любого блока устройства внешней памяти реально выполняется лишь обновление (или образование и наполнение) буфера кэша. Действительный обмен с устройством выполняется либо при выталкивании буфера вследствие замещения его содержимого, либо при выполнении специального системного вызова `sync` (или `fsync`), поддерживаемого специально для насильственного выталкивания во внешнюю память обновленных буферов кэша.

Эта традиционная схема буферизации вошла в противоречие с развитыми в современных вариантах ОС UNIX средствами управления виртуальной памятью и в особенности с механизмом отображения файлов в сегменты виртуальной памяти (см. пп. 2.4.5 и 3.1.2). (Мы не будем подробно объяснять здесь суть этих противоречий и предложим читателям поразмышлять над этим.) Поэтому в System V Release 4 появилась новая схема буферизации, пока используемая параллельно со старой схемой.

Суть новой схемы состоит в том, что на уровне ядра фактически воспроизводится механизм отображения файлов в сегменты виртуальной памяти. Во-первых, напомним о том, что ядро ОС UNIX действительно работает в собственной виртуальной памяти. Эта память имеет более сложную, но принципиально такую же структуру, что и пользовательская виртуальная память. Другими словами, виртуальная память ядра является сегментно-страничной, и наравне с виртуальной памятью пользовательских процессов поддерживается общей подсистемой управления виртуальной памятью. Из этого следует, во-вторых, что практически любая функция, обеспечиваемая ядром для пользователей, может быть обеспечена одними компонентами ядра для других его компонентов. В частности, это относится и к возможностям отображения файлов в сегменты виртуальной памяти.

Новая схема буферизации в ядре ОС UNIX главным образом основывается на том, что для организации буферизации можно не делать почти ничего специального. Когда один из пользовательских процессов открывает не открытый до этого времени файл, ядро образует новый сегмент и подключает к этому сегменту открываемый файл. После этого (независимо от того, будет ли пользовательский процесс работать с файлом в традиционном режиме с использованием системных вызовов `read` и `write` или подключит файл к сегменту своей виртуальной памяти) на уровне ядра работа будет производиться с тем ядерным сегментом, к которому подключен файл на уровне ядра. Основная идея нового подхода состоит в том, что устраняется разрыв между управлением виртуальной памятью и общесистемной буферизацией (это нужно было бы сделать давно, поскольку

очевидно, что основную буферизацию в операционной системе должен производить компонент управления виртуальной памятью).

Почему же нельзя отказаться от старого механизма буферизации? Все дело в том, что новая схема предполагает наличие некоторой непрерывной адресации внутри объекта внешней памяти (должен существовать изоморфизм между отображаемым и отображенным объектами). Однако, при организации файловых систем ОС UNIX достаточно сложно распределяет внешнюю память, что в особенности относится к i-узлам. Поэтому некоторые блоки внешней памяти приходится считать изолированными, и для них оказывается выгоднее использовать старую схему буферизации (хотя, возможно, в завтрашних вариантах UNIX и удастся полностью перейти к унифицированной новой схеме).

Системные вызовы для управления вводом/выводом

Для доступа (т.е. для получения возможности последующего выполнения операций ввода/вывода) к файлу любого вида (включая специальные файлы) пользовательский процесс должен выполнить предварительное подключение к файлу с помощью одного из системных вызовов `open`, `creat`, `dup` или `pipe`. Программные каналы и соответствующие системные вызовы мы рассмотрим в п. 3.4.3, а пока несколько более подробно, чем в п. 2.3.3, рассмотрим другие "инициализирующие" системные вызовы.

Последовательность действий системного вызова `open (pathname, mode)` следующая:

- анализируется непротиворечивость входных параметров (главным образом, относящихся к флагам режима доступа к файлу);
- выделяется или находится пространство для описателя файла в системной области данных процесса (u-области);
- в общесистемной области выделяется или находится существующее пространство для размещения системного описателя файла (структуры file);
- производится поиск в архиве файловой системы объекта с именем "pathname" и образуется или обнаруживается описатель файла уровня файловой системы (vnode в терминах UNIX V System 4);
- выполняется связывание vnode с ранее образованной структурой file.

Системные вызовы `open` и `creat` (почти) функционально эквивалентны. Любой существующий файл можно открыть с помощью системного вызова `creat`, и любой новый файл можно создать с помощью системного вызова `open`. Однако, применительно к системному вызову `creat` мы должны подчеркнуть, что в случае своего естественного применения (для создания файла) этот системный вызов создает новый элемент соответствующего каталога (в соответствии с заданным значением `pathname`), а также создает и соответствующим образом инициализирует новый i-узел.

Наконец, системный вызов `dup` (duplicate - скопировать) приводит к образованию нового дескриптора уже открытого файла. Этот специфический для ОС UNIX системный вызов служит исключительно для целей перенаправления ввода/вывода (см. п. 2.1.8). Его выполнение состоит в том, что в u-области системного пространства пользовательского процесса образуется новый описатель открытого файла, содержащий вновь образованный дескриптор файла (целое число), но ссылающийся на уже существующую общесистемную структуру file и содержащий те же самые признаки и флаги, которые соответствуют открытому файлу-образцу.

Другими важными системными вызовами являются системные вызовы `read` и `write`. Системный вызов `read` выполняется следующим образом:

- в общесистемной таблице файлов находится дескриптор указанного файла, и определяется, законно ли обращение от данного процесса к данному файлу в указанном режиме;
- на некоторое (короткое) время устанавливается синхронизационная блокировка на vnode данного файла (содержимое описателя не должно изменяться в критические моменты операции чтения);
- выполняется собственно чтение с использованием старого или нового механизма буферизации, после чего данные копируются, чтобы стать доступными в пользовательском адресном пространстве.

Операция записи выполняется аналогичным образом, но меняет содержимое буфера буферного пула.

Системный вызов `close` приводит к тому, что драйвер обрывает связь с соответствующим пользовательским процессом и (в случае последнего по времени закрытия устройства) устанавливает общесистемный флаг "драйвер свободен".

Наконец, для специальных файлов поддерживается еще один "специальный" системный вызов `ioctl`. Это единственный системный вызов, который обеспечивается для специальных файлов и не обеспечивается для остальных разновидностей файлов. Фактически, системный вызов `ioctl` позволяет произвольным образом расширить интерфейс любого драйвера. Параметры `ioctl` включают код операции и указатель на некоторую область памяти пользовательского процесса. Всю интерпретацию кода операции и соответствующих специфических параметров проводит драйвер.

Естественно, что поскольку драйверы главным образом предназначены для управления внешними устройствами, программный код драйвера должен содержать соответствующие средства для обработки прерываний от устройства. Вызов индивидуальной программы обработки прерываний в драйвере происходит из ядра операционной системы. Подобным же образом в драйвере может быть объявлен вход "timeout", к которому обращается ядро при истечении ранее заказанного драйвером времени (такой временной контроль является необходимым при управлении не слишком интеллектуальными устройствами).

Общая схема интерфейсной организации драйверов показана на рисунке 3.5. Как показывает этот рисунок, с точки зрения интерфейсов и общесистемного управления различаются два вида драйверов - символьные и блочные. С точки зрения внутренней организации выделяется еще один вид драйверов - потоковые (stream) драйверы (мы уже упоминали о потоках в п. 2.7.1). Однако по своему внешнему интерфейсу потоковые драйверы не отличаются от символьных.

Рис. 3.5. Интерфейсы и входные точки драйверов

Блочные драйверы

Блочные драйверы предназначаются для обслуживания внешних устройств с блочной структурой (магнитных дисков, лент и т.д.) и отличаются от прочих тем, что они

разрабатываются и выполняются с использованием системной буферизации. Другими словами, такие драйверы всегда работают через системный буферный пул. Как видно на рисунке 3.5, любое обращение к блочному драйверу для чтения или записи всегда проходит через предварительную обработку, которая заключается в попытке найти копию нужного блока в буферном пуле.

В случае, если копия требуемого блока не находится в буферном пуле или если по какой-либо причине требуется заменить содержимое некоторого обновленного буфера, ядро ОС UNIX обращается к процедуре `strategy` соответствующего блочного драйвера. `Strategy` обеспечивает стандартный интерфейс между ядром и драйвером. С использованием библиотечных подпрограмм, предназначенных для написания драйверов, процедура `strategy` может организовывать очереди обменов с устройством, например, с целью оптимизации движения магнитных головок на диске. Все обмены, выполняемые блочным драйвером, выполняются с буферной памятью. Перепись нужной информации в память соответствующего пользовательского процесса производится программами ядра, заведующими управлением буферами.

Символьные драйверы

Символьные драйверы главным образом предназначены для обслуживания устройств, обмены с которыми выполняются посимвольно, либо строками символов переменного размера. Типичным примером символьного устройства является простой принтер, принимающий один символ за один обмен.

Символьные драйверы не используют системную буферизацию. Они напрямую копируют данные из памяти пользовательского процесса при выполнении операций записи или в память пользовательского процесса при выполнении операций чтения, используя собственные буфера.

Следует отметить, что имеется возможность обеспечить символьный интерфейс для блочного устройства. В этом случае блочный драйвер использует дополнительные возможности процедуры `strategy`, позволяющие выполнять обмен без применения системной буферизации. Для драйвера, обладающего одновременно блочным и символьным интерфейсами, в файловой системе заводится два специальных файла, блочный и символьный. При каждом обращении драйвер получает информацию о том, в каком режиме он используется.

Потоковые драйверы

Как отмечалось в п. 2.7.1, основным назначением механизма потоков (streams) является повышение уровня модульности и гибкости драйверов со сложной внутренней логикой (более всего это относится к драйверам, реализующим развитые сетевые протоколы). Спецификой таких драйверов является то, что большая часть программного кода не зависит от особенностей аппаратного устройства. Более того, часто оказывается выгодно по-разному комбинировать части программного кода.

Все это привело к появлению потоковой архитектуры драйверов, которые представляют собой двунаправленный конвейер обрабатывающих модулей. В начале конвейера (ближе всего к пользовательскому процессу) находится заголовок потока, к которому прежде всего поступают обращения по инициативе пользователя. В конце конвейера (ближе всего к устройству) находится обычный драйвер устройства. В промежутке может

располагаться произвольное число обрабатывающих модулей, каждый из которых оформляется в соответствии с обязательным потоковым интерфейсом.

Взаимодействие процессов

Каждый процесс в ОС UNIX выполняется в собственной виртуальной памяти, т.е. если не предпринимать дополнительных усилий, то даже процессы-близнецы, образованные в результате выполнения системного вызова `fork()`, на самом деле полностью изолированы один от другого (если не считать того, что процесс-потомок наследует от процесса-предка все открытые файлы). Тем самым, в ранних вариантах ОС UNIX поддерживались весьма слабые возможности взаимодействия процессов, даже входящих в общую иерархию порождения (т.е. имеющих общего предка).

Очень слабые средства поддерживались и для взаимной синхронизации процессов. Практически, все ограничивалось возможностью реакции на сигналы, и наиболее распространенным видом синхронизации являлась реакция процесса-предка на сигнал о завершении процесса-потомка.

По-видимому, применение такого подхода являлось реакцией на чрезмерно сложные механизмы взаимодействия и синхронизации параллельных процессов, существовавшие в исторически предшествующей UNIX ОС Multics. Напомним (см. раздел 1.1), что в ОС Multics поддерживалась сегментно-страничная организация виртуальной памяти, и в общей виртуальной памяти могло выполняться несколько параллельных процессов, которые, естественно, могли взаимодействовать через общую память. За счет возможности включения одного и того же сегмента в разную виртуальную память аналогичная возможность взаимодействий существовала и для процессов, выполняемых не в общей виртуальной памяти.

Для синхронизации таких взаимодействий процессов поддерживался общий механизм семафоров, позволяющий, в частности, организовывать взаимное исключение процессов в критических участках их выполнения (например, при взаимно-исключающем доступе к разделяемой памяти). Этот стиль параллельного программирования в принципе обеспечивает большую гибкость и эффективность, но является очень трудным для использования. Часто в программах появляются трудно обнаруживаемые и редко воспроизводимые синхронизационные ошибки; использование явной синхронизации, не связанной неразрывно с теми объектами, доступ к которым синхронизуется, делает логику программ трудно постижимой, а текст программ - трудно читаемым.

Понятно, что стиль ранних вариантов ОС UNIX стимулировал существенно более простое программирование. В наиболее простых случаях процесс-потомок образовывался только для того, чтобы асинхронно с основным процессом выполнить какое-либо простое действие (например, запись в файл). В более сложных случаях процессы, связанные иерархией родства, создавали обрабатывающие "конвейеры" с использованием техники программных каналов (pipes). Эта техника особенно часто применяется при программировании на командных языках (см. раздел 5.2).

Долгое время отцы-основатели ОС UNIX считали, что в той области, для которой предназначался UNIX (разработка программного обеспечения, подготовка и сопровождение технической документации и т.д.) этих возможностей вполне достаточно. Однако постепенное распространение системы в других областях и сравнительная простота наращивания ее возможностей привели к тому, что со временем в разных вариантах ОС UNIX в совокупности появился явно избыточный набор системных средств,

предназначенных для обеспечения возможности взаимодействия и синхронизации процессов, которые не обязательно связаны отношением родства (в мире ОС UNIX эти средства обычно называют IPC от Inter-Process Communication Facilities). С появлением UNIX System V Release 4.0 (и более старшей версии 4.2) все эти средства были узаконены и вошли в фактический стандарт ОС UNIX современного образца.

Нельзя сказать, что средства IPC ОС UNIX идеальны хотя бы в каком-нибудь отношении. При разработке сложных асинхронных программных комплексов (например, систем реального времени) больше всего неудобств причиняет избыточность средств IPC. Всегда возможны несколько вариантов реализации, и очень часто невозможно найти критерии выбора. Дополнительную проблему создает тот факт, что в разных вариантах системы средства IPC реализуются по-разному, зачастую одни средства реализованы на основе использования других средств. Поэтому эффективность реализации различается, из-за чего усложняется разработка мобильных асинхронных программных комплексов.

Тем не менее, знать возможности IPC, безусловно, нужно, если относиться к ОС UNIX как к серьезной производственной операционной системе. В этом разделе мы рассмотрим основные стандартизованные возможности в основном на идейном уровне, не вдаваясь в технические детали.

Порядок рассмотрения не отражает какую-либо особую степень важности или предпочтительности конкретного средства. Мы начинаем с пакета средств IPC, которые появились в UNIX System V Release 3.0. Этот пакет включает:

- средства, обеспечивающие возможность наличия общей для процессов памяти (сегменты разделяемой памяти - shared memory segments);
- средства, обеспечивающие возможность синхронизации процессов при доступе к совместно используемым ресурсам, например, к разделяемой памяти (семафоры - semaphores);
- средства, обеспечивающие возможность посылки процессом сообщений другому произвольному процессу (очереди сообщений - message queues).

Эти механизмы объединяются в единый пакет, потому что соответствующие системные вызовы обладают близкими интерфейсами, а в их реализации используются многие общие подпрограммы. Вот основные общие свойства всех трех механизмов:

- Для каждого механизма поддерживается общесистемная таблица, элементы которой описывают всех существующих в данный момент представителей механизма (конкретные сегменты, семафоры или очереди сообщений).
- Элемент таблицы содержит некоторый числовой ключ, который является выбранным пользователем именем представителя соответствующего механизма. Другими словами, чтобы два или более процесса могли использовать некоторый механизм, они должны заранее договориться об именовании используемого представителя этого механизма и добиться того, чтобы тот же представитель не использовался другими процессами.
- Процесс, желающий начать пользоваться одним из механизмов, обращается к системе с системным вызовом из семейства "get", прямыми параметрами которого является ключ объекта и дополнительные флаги, а ответным параметром является числовой дескриптор, используемый в дальнейших системных вызовах подобно тому, как используется дескриптор файла при работе с файловой системой. Допускается использование специального значения ключа с символическим именем IPC_PRIVATE, обязывающего систему выделить новый элемент в таблице

соответствующего механизма независимо от наличия или отсутствия в ней элемента, содержащего то же значение ключа. При указании других значений ключа задание флага IPC_CREAT приводит к образованию нового элемента таблицы, если в таблице отсутствует элемент с указанным значением ключа, или нахождению элемента с этим значением ключа. Комбинация флагов IPC_CREAT и IPC_EXCL приводит к выдаче диагностики об ошибочной ситуации, если в таблице уже содержится элемент с указанным значением ключа.
- Защита доступа к ранее созданным элементам таблицы каждого механизма основывается на тех же принципах, что и защита доступа к файлам.

Перейдем к более детальному изучению конкретных механизмов этого семейства.

Разделяемая память

Для работы с разделяемой памятью используются четыре системных вызова:

- `shmget` создает новый сегмент разделяемой памяти или находит существующий сегмент с тем же ключом;
- `shmat` подключает сегмент с указанным дескриптором к виртуальной памяти обращающегося процесса;
- `shmdt` отключает от виртуальной памяти ранее подключенный к ней сегмент с указанным виртуальным адресом начала;
- наконец, системный вызов `shmctl` служит для управления разнообразными параметрами, связанными с существующим сегментом.

После того, как сегмент разделяемой памяти подключен к виртуальной памяти процесса, этот процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи, не прибегая к использованию дополнительных системных вызовов.

Синтаксис системного вызова `shmget` выглядит следующим образом:

```
shmid = shmget(key, size, flag);
```

Параметр `size` определяет желаемый размер сегмента в байтах. Далее работа происходит по общим правилам. Если в таблице разделяемой памяти находится элемент, содержащий заданный ключ, и права доступа не противоречат текущим характеристикам обращающегося процесса, то значением системного вызова является дескриптор существующего сегмента (и обратившийся процесс так и не узнает реального размера сегмента, хотя впоследствии его все-таки можно узнать с помощью системного вызова `shmctl`). В противном случае создается новый сегмент с размером не меньше установленного в системе минимального размера сегмента разделяемой памяти и не больше установленного максимального размера. Создание сегмента не означает немедленного выделения под него основной памяти. Это действие откладывается до выполнения первого системного вызова подключения сегмента к виртуальной памяти некоторого процесса. Аналогично, при выполнении последнего системного вызова отключения сегмента от виртуальной памяти соответствующая основная память освобождается.

Подключение сегмента к виртуальной памяти выполняется путем обращения к системному вызову `shmat`:

```
virtaddr = shmat(id, addr, flags);
```

Здесь `id` - это ранее полученный дескриптор сегмента, а `addr` - желаемый процессом виртуальный адрес, который должен соответствовать началу сегмента в виртуальной памяти. Значением системного вызова является реальный виртуальный адрес начала сегмента (его значение не обязательно совпадает со значением прямого параметра `addr`). Если значением `addr` является нуль, ядро выбирает наиболее удобный виртуальный адрес начала сегмента. Кроме того, ядро старается обеспечить (но не гарантирует) выбор такого стартового виртуального адреса сегмента, который обеспечивал бы отсутствие перекрывающихся виртуальных адресов данного разделяемого сегмента, сегмента данных и сегмента стека процесса (два последних сегмента могут расширяться).

Для отключения сегмента от виртуальной памяти используется системный вызов `shmdt`:

```
shmdt(addr);
```

где `addr` - это виртуальный адрес начала сегмента в виртуальной памяти, ранее полученный от системного вызова `shmat`. Естественно, система гарантирует (на основе использования таблицы сегментов процесса), что указанный виртуальный адрес действительно является адресом начала (разделяемого) сегмента в виртуальной памяти данного процесса.

Системный вызов `shmctl`:

```
shmctl(id, cmd, shsstatbuf);
```

содержит прямой параметр `cmd`, идентифицирующий требуемое конкретное действие, и предназначен для выполнения различных функций. Видимо, наиболее важной является функция уничтожения сегмента разделяемой памяти. Уничтожение сегмента производится следующим образом. Если к моменту выполнения системного вызова ни один процесс не подключил сегмент к своей виртуальной памяти, то основная память, занимаемая сегментом, освобождается, а соответствующий элемент таблицы разделяемых сегментов объявляется свободным. В противном случае в элементе таблицы сегментов выставляется флаг, запрещающий выполнение системного вызова `shmget` по отношению к этому сегменту, но процессам, успевшим получить дескриптор сегмента, по-прежнему разрешается подключать сегмент к своей виртуальной памяти. При выполнении последнего системного вызова отключения сегмента от виртуальной памяти операция уничтожения сегмента завершается.

Семафоры

Механизм семафоров, реализованный в ОС UNIX, является обобщением классического механизма семафоров общего вида, предложенного более 25 лет тому назад известным голландским специалистом профессором Дейкстрой. Заметим, что целесообразность введения такого обобщения достаточно сомнительна. Обычно наоборот использовался облегченный вариант семафоров Дейкстры - так называемые двоичные семафоры. Мы не будем здесь углубляться в общую теорию синхронизации на основе семафоров, но заметим, что достаточность в общем случае двоичных семафоров доказана (известен алгоритм реализации семафоров общего вида на основе двоичных). Конечно, аналогичные рассуждения можно было бы применить и к варианту семафоров, примененному в ОС UNIX.

Семафор в ОС UNIX состоит из следующих элементов:

- значение семафора;
- идентификатор процесса, который хронологически последним работал с семафором;
- число процессов, ожидающих увеличения значения семафора;
- число процессов, ожидающих нулевого значения семафора.

Для работы с семафорами поддерживаются три системных вызова:

- `semget` для создания и получения доступа к набору семафоров;
- `semop` для манипулирования значениями семафоров (это именно тот системный вызов, который позволяет процессам синхронизоваться на основе использования семафоров);
- `semctl` для выполнения разнообразных управляющих операций над набором семафоров.

Системный вызов `semget` имеет следующий синтаксис:

```
id = semget(key, count, flag);
```

где прямые параметры `key` и `flag` и возвращаемое значение системного вызова имеют тот же смысл, что для других системных вызовов семейства "get", а параметр `count` задает число семафоров в наборе семафоров, обладающих одним и тем же ключом. После этого индивидуальный семафор идентифицируется дескриптором набора семафоров и номером семафора в этом наборе. Если к моменту выполнения системного вызова `semget` набор семафоров с указанным ключом уже существует, то обращающийся процесс получит соответствующий дескриптор, но так и не узнает о реальном числе семафоров в группе (хотя позже это все-таки можно узнать с помощью системного вызова `semctl`).

Основным системным вызовом для манипулирования семафором является `semop`:

```
oldval = semop(id, oplist, count);
```

где `id` - это ранее полученный дескриптор группы семафоров, `oplist` - массив описателей операций над семафорами группы, а `count` - размер этого массива. Значение, возвращаемое системным вызовом, является значением последнего обработанного семафора. Каждый элемент массива `oplist` имеет следующую структуру:

- номер семафора в указанном наборе семафоров;
- операция;
- флаги.

Если проверка прав доступа проходит нормально, и указанные в массиве `oplist` номера семафоров не выходят за пределы общего размера набора семафоров, то системный вызов выполняется следующим образом. Для каждого элемента массива `oplist` значение соответствующего семафора изменяется в соответствии со значением поля "операция".

- Если значение поля операции положительно, то значение семафора увеличивается на единицу, а все процессы, ожидающие увеличения значения семафора, активизируются (пробуждаются в терминологии UNIX).

- Если значение поля операции равно нулю, то если значение семафора также равно нулю, выбирается следующий элемент массива `oplist`. Если же значение семафора отлично от нуля, то ядро увеличивает на единицу число процессов, ожидающих нулевого значения семафора, а обратившийся процесс переводится в состояние ожидания (усыпляется в терминологии UNIX).
- Наконец, если значение поля операции отрицательно, и его абсолютное значение меньше или равно значению семафора, то ядро прибавляет это отрицательное значение к значению семафора. Если в результате значение семафора стало нулевым, то ядро активизирует (пробуждает) все процессы, ожидающие нулевого значения этого семафора. Если же значение семафора меньше абсолютной величины поля операции, то ядро увеличивает на единицу число процессов, ожидающих увеличения значения семафора и откладывает (усыпляет) текущий процесс до наступления этого события.

Основным поводом для введения массовых операций над семафорами было стремление дать программистам возможность избегать тупиковых ситуаций в связи с семафорной синхронизацией. Это обеспечивается тем, что системный вызов `semop`, каким бы длинным он не был (по причине потенциально неограниченной длины массива `oplist`) выполняется как атомарная операция, т.е. во время выполнения `semop` ни один другой процесс не может изменить значение какого-либо семафора.

Наконец, среди флагов-параметров системного вызова `semop` может содержаться флаг с символическим именем IPC_NOWAIT, наличие которого заставляет ядро ОС UNIX не блокировать текущий процесс, а лишь сообщать в ответных параметрах о возникновении ситуации, приведшей бы к блокированию процесса при отсутствии флага IPC_NOWAIT. Мы не будем обсуждать здесь возможности корректного завершения работы с семафорами при незапланированном завершении процесса; заметим только, что такие возможности обеспечиваются.

Системный вызов `semctl` имеет формат

```
semctl(id, number, cmd, arg);
```

где `id` - это дескриптор группы семафоров, `number` - номер семафора в группе, `cmd` - код операции, а `arg` - указатель на структуру, содержимое которой интерпретируется по-разному, в зависимости от операции. В частности, с помощью `semctl` можно уничтожить индивидуальный семафор в указанной группе. Однако детали этого системного вызова настолько громоздки, что мы рекомендуем в случае необходимости обращаться к технической документации используемого варианта операционной системы.

Очереди сообщений

Для обеспечения возможности обмена сообщениями между процессами этот механизм поддерживается следующими системными вызовами:

- `msgget` для образования новой очереди сообщений или получения дескриптора существующей очереди;
- `msgsnd` для посылки сообщения (вернее, для его постановки в указанную очередь сообщений);
- `msgrcv` для приема сообщения (вернее, для выборки сообщения из очереди сообщений);
- `msgctl` для выполнения ряда управляющих действий.

Системный вызов `msgget` обладает стандартным для семейства "get" системных вызовов синтаксисом:

```
msgqid = msgget(key, flag);
```

Ядро хранит сообщения в виде связного списка (очереди), а дескриптор очереди сообщений является индексом в массиве заголовков очередей сообщений. В дополнение к информации, общей для всех механизмов IPC в UNIX System V, в заголовке очереди хранятся также:

- указатели на первое и последнее сообщение в данной очереди;
- число сообщений и общее количество байтов данных во всех них вместе взятых;
- идентификаторы процессов, которые последними послали или приняли сообщение через данную очередь;
- временные метки последних выполненных операций `msgsnd`, `msgrsv` и `msgctl`.

Как обычно, при выполнении системного вызова `msgget` ядро ОС UNIX либо создает новую очередь сообщений, помещая ее заголовок в таблицу очередей сообщений и возвращая пользователю дескриптор вновь созданной очереди, либо находит элемент таблицы очередей сообщений, содержащий указанный ключ, и возвращает соответствующий дескриптор очереди. На рисунке 3.6 показаны структуры данных, используемые для организации очередей сообщений.

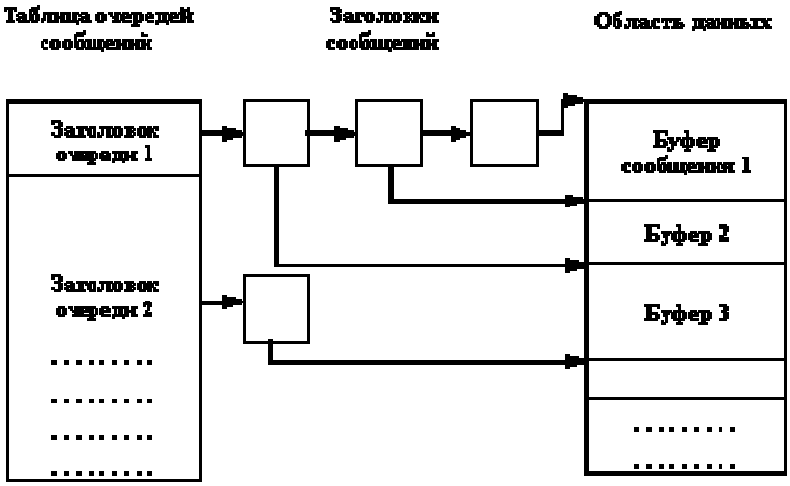

Рис. 3.6. Структуры данных, используемые для организации очередей сообщений

Для посылки сообщения используется системный вызов `msgsnd`:

```
msgsnd(msgqid, msg, count, flag);
```

где `msg` - это указатель на структуру, содержащую определяемый пользователем целочисленный тип сообщения и символьный массив - собственно сообщение; `count` задает размер сообщения в байтах, а `flag` определяет действия ядра при выходе за пределы допустимых размеров внутренней буферной памяти.

Для того, чтобы ядро успешно поставило указанное сообщение в указанную очередь сообщений, должны быть выполнены следующие условия: обращающийся процесс должен иметь соответствующие права по записи в данную очередь сообщений; длина

сообщения не должна превосходить установленный в системе верхний предел; общая длина сообщений (включая вновь посылаемое) не должна превосходить установленный предел; указанный в сообщении тип сообщения должен быть положительным целым числом. В этом случае обратившийся процесс успешно продолжает свое выполнение, оставив отправленное сообщение в буфере очереди сообщений. Тогда ядро активизирует (пробуждает) все процессы, ожидающие поступления сообщений из данной очереди.

Если же оказывается, что новое сообщение невозможно буферизовать в ядре по причине превышения верхнего предела суммарной длины сообщений, находящихся в одной очереди сообщений, то обратившийся процесс откладывается (усыпляется) до тех пор, пока очередь сообщений не разгрузится процессами, ожидающими получения сообщений. Чтобы избежать такого откладывания, обращающийся процесс должен указать в числе параметров системного вызова `msgsnd` значение флага с символическим именем IPC_NOWAIT (как в случае использования семафоров), чтобы ядро выдало свидетельствующий об ошибке код возврата системного вызова `mgdsng` в случае невозможности включить сообщение в указанную очередь.

Для приема сообщения используется системный вызов `msgrcv`:

```
count = msgrcv(id, msg, maxcount, type, flag);
```

Здесь `msg` - это указатель на структуру данных в адресном пространстве пользователя, предназначенную для размещения принятого сообщения; `maxcount` задает размер области данных (массива байтов) в структуре `msg`; значение `type` специфицирует тип сообщения, которое желательно принять; значение параметра `flag` указывает ядру, что следует предпринять, если в указанной очереди сообщений отсутствует сообщение с указанным типом. Возвращаемое значение системного вызова задает реальное число байтов, переданных пользователю.

Выполнение системного вызова, как обычно, начинается с проверки правомочности доступа обращающегося процесса к указанной очереди. Далее, если значением параметра `type` является нуль, ядро выбирает первое сообщение из указанной очереди сообщений и копирует его в заданную пользовательскую структуру данных. После этого корректируется информация, содержащаяся в заголовке очереди (число сообщений, суммарный размер и т.д.). Если какие-либо процессы были отложены по причине переполнения очереди сообщений, то все они активизируются. В случае, если значение параметра `maxcount` оказывается меньше реального размера сообщения, ядро не удаляет сообщение из очереди и возвращает код ошибки. Однако, если задан флаг MSG_NOERROR, то выборка сообщения производится, и в буфер пользователя переписываются первые `maxcount` байтов сообщения.

Путем задания соответствующего значения параметра `type` пользовательский процесс может потребовать выборки сообщения некоторого конкретного типа. Если это значение является положительным целым числом, ядро выбирает из очереди сообщений первое сообщение с таким же типом. Если же значение параметра `type` есть отрицательное целое число, то ядро выбирает из очереди первое сообщение, значение типа которого меньше или равно абсолютному значению параметра `type`.

Во всех случаях, если в указанной очереди отсутствуют сообщения, соответствующие спецификации параметра `type`, ядро откладывает (усыпляет) обратившийся процесс до появления в очереди требуемого сообщения. Однако, если в параметре `flag` задано

значение флага IPC_NOWAIT, то процесс немедленно оповещается об отсутствии сообщения в очереди путем возврата кода ошибки.

Системный вызов

```
msgctl(id, cmd, mstatbuf);
```

служит для опроса состояния описателя очереди сообщений, изменения его состояния (например, изменения прав доступа к очереди) и для уничтожения указанной очереди сообщений (детали мы опускаем).

Программные каналы

Как мы уже неоднократно отмечали, традиционным средством взаимодействия и синхронизации процессов в ОС UNIX являются программные каналы (pipes). Теоретически программный канал позволяет взаимодействовать любому числу процессов, обеспечивая дисциплину FIFO (first-in-first-out). Другими словами, процесс, читающий из программного канала, прочитает те данные, которые были записаны в программный канал наиболее давно. В традиционной реализации программных каналов для хранения данных использовались файлы. В современных версиях ОС UNIX для реализации программных каналов применяются другие средства IPC (в частности, очереди сообщений).

Различаются два вида программных каналов - именованные и неименованные. Именованный программный канал может служить для общения и синхронизации произвольных процессов, знающих имя данного программного канала и имеющих соответствующие права доступа. Неименованным программным каналом могут пользоваться только создавший его процесс и его потомки (необязательно прямые).

Для создания именованного программного канала (или получения к нему доступа) используется обычный файловый системный вызов `open`. Для создания же неименованного программного канала существует специальный системный вызов `pipe` (исторически более ранний). Однако после получения соответствующих дескрипторов оба вида программных каналов используются единообразно с помощью стандартных файловых системных вызовов `read`, `write` и `close`.

Системный вызов `pipe` имеет следующий синтаксис:

```
pipe(fdptr);
```

где `fdptr` - это указатель на массив из двух целых чисел, в который после создания неименованного программного канала будут помещены дескрипторы, предназначенные для чтения из программного канала (с помощью системного вызова `read`) и записи в программный канал (с помощью системного вызова `write`). Дескрипторы неименованного программного канала - это обычные дескрипторы файлов, т.е. такому программному каналу соответствуют два элемента таблицы открытых файлов процесса. Поэтому при последующем использовании системных вызовов `read` и `write` процесс совершенно не обязан отличать случай использования программных каналов от случая использования обычных файлов (собственно, на этом и основана идея перенаправления ввода/вывода и организации конвейеров).

Для создания именованных программных каналов (или получения доступа к уже существующим каналам) используется обычный системный вызов `open`. Основным

отличием от случая открытия обычного файла является то, что если именованный программный канал открывается на запись, и ни один процесс не открыл тот же программный канал для чтения, то обращающийся процесс блокируется (усыпляется) до тех пор, пока некоторый процесс не откроет данный программный канал для чтения (аналогично обрабатывается открытие для чтения). Повод для использования такого режима работы состоит в том, что, вообще говоря, бессмысленно давать доступ к программному каналу на чтение (запись) до тех пор, пока некоторый другой процесс не обнаружит готовности писать в данный программный канал (соответственно читать из него). Понятно, что если бы эта схема была абсолютной, то ни один процесс не смог бы начать работать с заданным именованным программным каналом (кто-то должен быть первым). Поэтому в числе флагов системного вызова `open` имеется флаг NO_DELAY, задание которого приводит к тому, что именованный программный канал открывается независимо от наличия соответствующего партнера.

Запись данных в программный канал и чтение данных из программного канала (независимо от того, именованный он или неименованный) выполняются с помощью системных вызовов `read` и `write`. Отличие от случая использования обычных файлов состоит лишь в том, что при записи данные помещаются в начало канала, а при чтении выбираются (освобождая соответствующую область памяти) из конца канала. Как всегда, возможны ситуации, когда при попытке записи в программный канал оказывается, что канал переполнен, и тогда обращающийся процесс блокируется, пока канал не разгрузится (если только не указан флаг нежелательности блокировки в числе параметров системного вызова `write`), или когда при попытке чтения из программного канала оказывается, что канал пуст (или в нем отсутствует требуемое количество байтов информации), и тогда обращающийся процесс блокируется, пока канал не загрузится соответствующим образом (если только не указан флаг нежелательности блокировки в числе параметров системного вызова `read`).

Окончание работы процесса с программным каналом (независимо от того, именованный он или неименованный) производится с помощью системного вызова `close`. В основном, действия ядра при закрытии программного канала аналогичны действиям при закрытии обычного файла. Однако имеется отличие в том, что при выполнении последнего закрытия канала по записи все процессы, ожидающие чтения из программного канала (т.е. процессы, обратившиеся к ядру с системным вызовом `read` и отложенные по причине недостатка данных в канале), активизируются с возвратом кода ошибки из системного вызова. (Это совершенно оправданно в случае неименованных программных каналов: если достоверно известно, что больше нечего читать, то зачем заставлять далее ждать чтения. Для именованных программных каналов это решение не является очевидным, но соответствует общей политике ОС UNIX о раннем предупреждении процессов.)

Программные гнезда (sockets)

Операционная система UNIX с самого начала проектировалась как сетевая ОС в том смысле, что должна была обеспечивать явную возможность взаимодействия процессов, выполняющихся на разных компьютерах, соединенных сетью передачи данных. Главным образом, эта возможность базировалась на обеспечении файлового интерфейса для устройств (включая сетевые адаптеры) на основе понятия специального файла. Другими словами, два или более процессов, располагающихся на разных компьютерах, могли договориться о способе взаимодействия на основе использования возможностей соответствующих сетевых драйверов.

Эти базовые возможности были в принципе достаточными для создания сетевых утилит; в частности, на их основе был создан исходный в ОС UNIX механизм сетевых взаимодействий `uucp`. Однако организация сетевых взаимодействий пользовательских процессов была затруднительна главным образом потому, что при использовании конкретной сетевой аппаратуры и конкретного сетевого протокола требовалось выполнять множество системных вызовов `ioctl`, что делало программы зависимыми от специфической сетевой среды. Требовался поддерживаемый ядром механизм, позволяющий скрыть особенности этой среды и позволить единообразно взаимодействовать процессам, выполняющимся на одном компьютере, в пределах одной локальной сети или разнесенным на разные компьютеры территориально распределенной сети. Первое решение этой проблемы было предложено и реализовано в UNIX BSD 4.1 в 1982 г. (вводную информацию см. в п. 2.7.3).

На уровне ядра механизм программных гнезд поддерживается тремя составляющими: компонентом уровня программных гнезд (независящим от сетевого протокола и среды передачи данных), компонентом протокольного уровня (независящим от среды передачи данных) и компонентом уровня управления сетевым устройством (см. рисунок 3.7).

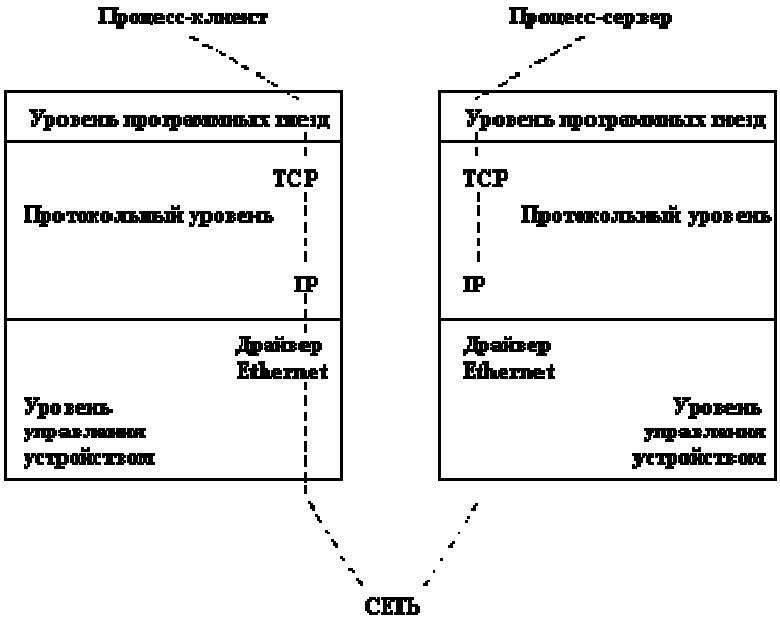

Рис. 3.7. Одна из возможных конфигураций программных гнезд

Допустимые комбинации протоколов и драйверов задаются при конфигурации системы, и во время работы системы их менять нельзя. Легко видеть, что по своему духу организация программных гнезд близка к идее потоков (см. пп. 2.7.1 и 3.4.6), поскольку основана на разделении функций физического управления устройством, протокольных функций и функций интерфейса с пользователями. Однако это менее гибкая схема, поскольку не допускает изменения конфигурации "на ходу".

Взаимодействие процессов на основе программных гнезд основано на модели "клиент-сервер". Процесс-сервер "слушает (listens)" свое программное гнездо, одну из конечных точек двунаправленного пути коммуникаций, а процесс-клиент пытается общаться с процессом-сервером через другое программное гнездо, являющееся второй конечной точкой коммуникационного пути и, возможно, располагающееся на другом компьютере. Ядро поддерживает внутренние соединения и маршрутизацию данных от клиента к серверу.

Программные гнезда с общими коммуникационными свойствами, такими как способ именования и протокольный формат адреса, группируются в домены. Наиболее часто используемыми являются "домен системы UNIX" для процессов, которые взаимодействуют через программные гнезда в пределах одного компьютера, и "домен Internet" для процессов, которые взаимодействуют в сети в соответствии с семейством протоколов TCP/IP (см. п. 2.7.2).

Выделяются два типа программных гнезд - гнезда с виртуальным соединением (в начальной терминологии stream sockets) и датаграммные гнезда (datagram sockets). При использовании программных гнезд с виртуальным соединением обеспечивается передача данных от клиента к серверу в виде непрерывного потока байтов с гарантией доставки. При этом до начала передачи данных должно быть установлено соединение, которое поддерживается до конца коммуникационной сессии. Датаграммные программные гнезда не гарантируют абсолютной надежной, последовательной доставки сообщений и отсутствия дубликатов пакетов данных - датаграмм. Но для использования датаграммного режима не требуется предварительное дорогостоящее установление соединений, и поэтому этот режим во многих случаях является предпочтительным. Система по умолчанию сама обеспечивает подходящий протокол для каждой допустимой комбинации "домен-гнездо". Например, протокол TCP используется по умолчанию для виртуальных соединений, а протокол UDP - для датаграммного способа коммуникаций (информация об этих протоколах представлена в п. 2.7.2).

Для работы с программными гнездами поддерживается набор специальных библиотечных функций (в UNIX BSD это системные вызовы, однако, как мы отмечали в п. 2.7.3, в UNIX System V они реализованы на основе потокового интерфейса TLI). Рассмотрим кратко интерфейсы и семантику этих функций.

Для создания нового программного гнезда используется функция `socket`:

```
sd = socket(domain, type, protocol);
```

где значение параметра `domain` определяет домен данного гнезда, параметр `type` указывает тип создаваемого программного гнезда (с виртуальным соединением или датаграммное), а значение параметра `protocol` определяет желаемый сетевой протокол. Заметим, что если значением параметра `protocol` является нуль, то система сама выбирает подходящий протокол для комбинации значений параметров `domain` и `type`, это наиболее распространенный способ использования функции `socket`. Возвращаемое функцией значение является дескриптором программного гнезда и используется во всех последующих функциях. Вызов функции `close(sd)` приводит к закрытию (уничтожению) указанного программного гнезда.

Для связывания ранее созданного программного гнезда с именем используется функция `bind`:

```
bind(sd, socknm, socknlen);
```

Здесь `sd` - дескриптор ранее созданного программного гнезда, `socknm` - адрес структуры, которая содержит имя (идентификатор) гнезда, соответствующее требованиям домена данного гнезда и используемого протокола (в частности, для домена системы UNIX имя является именем объекта в файловой системе, и при создании программного гнезда действительно создается файл), параметр `socknlen` содержит длину в байтах структуры

socknm (этот параметр необходим, поскольку длина имени может весьма различаться для разных комбинаций "домен-протокол").

С помощью функции connect процесс-клиент запрашивает систему связаться с существующим программным гнездом (у процесса-сервера):

```
connect(sd, socknm, socknlen);
```

Смысл параметров такой же, как у функции bind, однако в качестве имени указывается имя программного гнезда, которое должно находиться на другой стороне коммуникационного канала. Для нормального выполнения функции необходимо, чтобы у гнезда с дескриптором sd и у гнезда с именем socknm были одинаковые домен и протокол. Если тип гнезда с дескриптором sd является датаграммным, то функция connect служит только для информирования системы об адресе назначения пакетов, которые в дальнейшем будут посылаться с помощью функции send; никакие действия по установлению соединения в этом случае не производятся.

Функция listen предназначена для информирования системы о том, что процесс-сервер планирует установление виртуальных соединений через указанное гнездо:

```
listen(sd, qlength);
```

Здесь sd - это дескриптор существующего программного гнезда, а значением параметра qlength является максимальная длина очереди запросов на установление соединения, которые должны буферизоваться системой, пока их не выберет процесс-сервер.

Для выборки процессом-сервером очередного запроса на установление соединения с указанным программным гнездом служит функция accept:

```
nsd = accept(sd, address, addrlen);
```

Параметр sd задает дескриптор существующего программного гнезда, для которого ранее была выполнена функция listen; address указывает на массив данных, в который должна быть помещена информация, характеризующая имя программного гнезда клиента, со стороны которого поступает запрос на установление соединения; addrlen - адрес, по которому находится длина массива address. Если к моменту выполнения функции accept очередь запросов на установление соединений пуста, то процесс-сервер откладывается до поступления запроса. Выполнение функции приводит к установлению виртуального соединения, а ее значением является новый дескриптор программного гнезда, который должен использоваться при работе через данное соединение. По адресу addrlen помещается реальный размер массива данных, которые записаны по адресу address. Процесс-сервер может продолжать "слушать" следующие запросы на установление соединения, пользуясь установленным соединением.

Для передачи и приема данных через программные гнезда с установленным виртуальным соединением используются функции send и recv:

```
count = send(sd, msg, length, flags);

count = recv(sd, buf, length, flags);
```

В функции `send` параметр `sd` задает дескриптор существующего программного гнезда с установленным соединением; `msg` указывает на буфер с данными, которые требуется послать; `length` задает длину этого буфера. Наиболее полезным допустимым значением параметра `flags` является значение с символическим именем MSG_OOB, задание которого означает потребность во внеочередной посылке данных. "Внеочередные" сообщения посылаются помимо нормального для данного соединения потока данных, обгоняя все непрочитанные сообщения. Потенциальный получатель данных может получить специальный сигнал и в ходе его обработки немедленно прочитать внеочередные данные. Возвращаемое значение функции равняется числу реально посланных байтов и в нормальных ситуациях совпадает со значением параметра `length`.

В функции `recv` параметр `sd` задает дескриптор существующего программного гнезда с установленным соединением; `buf` указывает на буфер, в который следует поместить принимаемые данные; `length` задает максимальную длину этого буфера. Наиболее полезным допустимым значением параметра `flags` является значение с символическим именем MSG_PEEK, задание которого приводит к переписи сообщения в пользовательский буфер без его удаления из системных буферов. Возвращаемое значение функции является числом байтов, реально помещенных в `buf`.

Заметим, что в случае использования программных гнезд с виртуальным соединением вместо функций `send` и `recv` можно использовать обычные файловые системные вызовы `read` и `write`. Для программных гнезд они выполняются абсолютно аналогично функциям `send` и `recv`. Это позволяет создавать программы, не зависящие от того, работают ли они с обычными файлами, программными каналами или программными гнездами.

Для посылки и приема сообщений в датаграммном режиме используются функции `sendto` и `recvfrom`:

```
count = sendto(sd, msg, length, flags, socknm, socknlen);

count = recvfrom(sd, buf, length, flags, socknm, socknlen);
```

Смысл параметров `sd`, `msg`, `buf` и `lenght` аналогичен смыслу одноименных параметров функций `send` и `recv`. Параметры `socknm` и `socknlen` функции `sendto` задают имя программного гнезда, в которое посылается сообщение, и могут быть опущены, если до этого вызывалась функция `connect`. Параметры `socknm` и `socknlen` функции `recvfrom` позволяют серверу получить имя пославшего сообщение процесса-клиента.

Наконец, для немедленной ликвидации установленного соединения используется системный вызов `shutdown`:

```
shutdown(sd, mode);
```

Вызов этой функции означает, что нужно немедленно остановить коммуникации либо со стороны посылающего процесса, либо со стороны принимающего процесса, либо с обеих сторон (в зависимости от значения параметра `mode`). Действия функции `shutdown` отличаются от действий функции `close` тем, что, во-первых, выполнение последней "притормаживается" до окончания попыток системы доставить уже отправленные сообщения. Во-вторых, функция `shutdown` разрывает соединение, но не ликвидирует дескрипторы ранее соединенных гнезд. Для окончательной их ликвидации все равно требуется вызов функции `close`.

Замечание: приведенная в этом пункте информация может несколько отличаться от требований реально используемой вами системы. В основном это относится к символическим именам констант. Постоянная беда пользователей ОС UNIX состоит в том, что от версии к версии меняются символические имена и имена системных структурных типов.

Потоки (streams)

Здесь нам почти нечего добавить к материалу, приведенному в п. 2.7.1. На основе использования механизма потоковых сетевых драйверов в UNIX System V создана библиотека TLI (Transport Layer Interface), обеспечивающая транспортный сервис на основе стека протоколов TCP/IP. Возможности этого пакета превышают описанные выше возможности программных гнезд и, конечно, позволяют организовывать разнообразные виды коммуникации процессов. Однако многообразие и сложность набора функций библиотеки TLI не позволяют нам в подробностях описать их в рамках этого курса. Кроме того, TLI относится, скорее, не к теме ОС UNIX, а к теме реализаций семиуровневой модели ISO/OSI. Поэтому в случае необходимости мы рекомендуем пользоваться технической документацией по используемому варианту ОС UNIX или читать специальную литературу, посвященную сетевым возможностям современных версий UNIX.

Мобильное программирование в среде ОС UNIX

Одним из основных преимуществ семейства операционных систем типа UNIX и возникшего на их основе подхода к стандартизации интерфейсов операционных систем (важная часть общего подхода открытых систем) является то, что они обеспечивают единую операционную среду на компьютерах с разной архитектурой. Конечно, в начальном периоде истории ОС UNIX эта единообразность операционной среды являлась следствием мобильности единого текстового варианта системы. Когда начали появляться варианты ОС UNIX с разными исходными текстами, единообразность операционной среды стала нарушаться. И разработчикам ОС, и поставщикам аппаратных и программных средств было понятно, что складывающаяся ситуация наносит урон и производителям, и пользователям. Вместе с тем, уже нельзя было надеяться, что когда-нибудь удастся вернуться к единой реализации ОС UNIX. Выход был найден на пути стандартизации интерфейсов и семантики программных средств разного уровня, которые должна поддерживать любая операционная система, претендующая на операционную совместимость с "ОС UNIX". (Некоторые детали современного состояния процесса стандартизации излагаются в разделе 7.5.)

Конечно и до сих пор в разных реализациях ОС UNIX операционные среды несколько отличаются. Иногда (и довольно часто) бывает так, что утрачивается операционная совместимость даже при выпуске новой версии ОС. Но тем не менее, можно говорить о некотором общем подмножестве операционных средств, которые полностью стандартизованы и должны поддерживаться любым современным вариантом ОС UNIX. Этого подмножества оказывается достаточно для создания широкого класса мобильных приложений (хотя, конечно, некоторые особо сложные приложения, в особенности, связанные с реальным временем, пока удается делать мобильными только при использовании единой реализации ОС UNIX). В этой части курса мы рассмотрим основные приемы прикладного мобильного программирования в среде ОС UNIX, неявно подразумевая использование языка Си.

Замечание: Язык Си был и остается основным инструментом мобильного программирования в среде UNIX-систем. Многие считают (и мы с этим согласны), что более удобно, эффективно и надежно использовать языки объектно-ориентированного программирования, среди которых в настоящее время наиболее распространен язык Си++. Однако поддержка мобильного программирования на Си++ пока гораздо слабее, чем в случае Си (если, конечно, не ограничиваться использованием подмножества Си языка Си++ - но какой в этом смысл?). Можно, правда, надеяться, что после принятия летом 1995 г. международного стандарта языка Си++, который включает стандарты наиболее важных библиотек классов, через некоторое время такая поддержка появится.

Стандартные библиотеки

Очевидным требованием к операционной среде, поддерживающей мобильное прикладное программирование, является то, что все функции, предоставляемые ею прикладной программе, должны быть четко специфицированы и должны точно соответствовать этим спецификациям в любой реализации операционной среды. В UNIX-ориентированных средах это требование удовлетворяется за счет наличия нескольких стандартизованных библиотек функций и соответствующих наборов файлов заголовков (header-файлов).

Библиотека системных вызовов

Базовой библиотекой любого варианта системы UNIX является библиотека системных вызовов. Сейчас невозможно найти два варианта ОС UNIX с разными названиями, наборы системных вызовов которых полностью бы совпадали. Однако, любой такой вариант поддерживает системные вызовы, которые специфицированы в стандартах, упоминаемых в разделе 7.5. К полностью стандартным системным вызовам относятся системные вызовы для работы с файлами (включая специальные файлы), системные вызовы для управления процессами (`fork` и семейство `exec`), системные вызовы класса IPC (хотя, как мы упоминали в п. 3.5.4, в UNIX System V механизм программных каналов реализован не в виде набора системных вызовов ядра ОС, а как набор библиотечных функций над пакетом TLI). Приведенное в скобках замечание на самом деле является очень важным. Пользователя, стремящегося создать мобильное приложение с использованием системных вызовов, не должны волновать детали реализации. Важно, чтобы состав системных вызовов, их интерфейсы и семантика соответствовали стандартам.

Теперь мы можем сформулировать правило прикладного мобильного программирования с использованием системных вызовов:

Проектируя и разрабатывая прикладную систему, убедитесь, что вы не используете системные вызовы, не входящие в стандарт.

Придерживаясь этого правила, с большой вероятностью вы не будете иметь проблем с переносом программы в среду другого варианта ОС UNIX по причине несовместимости наборов системных вызовов.

Библиотека ввода/вывода

Традиционной для ОС UNIX библиотекой функций более высокого уровня, чем библиотека системных вызовов, является, так называемая, стандартная библиотека ввода/вывода (stdio). Основной набор функций этой библиотеки служит для выполнения файловых операций с буферизацией данных в памяти пользовательского процесса. Библиотека ввода/вывода фактически стандартизована очень давно, и ей можно безопасно

пользоваться в любой операционной среде. В частности, единообразные библиотеки ввода/вывода поддерживаются во всех современных реализациях системы программирования языка Си, выполненных не в среде ОС UNIX (включая реализации в среде MS-DOS).

Поэтому можно сформулировать правило мобильного программирования с использованием библиотеки ввода/вывода:

Если для разрабатываемой вами прикладной программы достаточно возможностей библиотеки ввода/вывода, ограничьтесь использованием этой библиотеки.

Придерживаясь этого правила, с большой вероятностью вы не будете иметь проблем, связанных с вводом/выводом, при переносе вашей программы в любую операционную среду (не обязательно UNIX-ориентированную), в которой поддерживается стандартная библиотека ввода/вывода.

Дополнительные библиотеки

Понятно, что при прикладном программировании используются не только библиотеки системных вызовов и ввода/вывода. Существует масса других библиотечных функций, предназначенных, например, для разнообразных преобразований форматов данных, математических вычислений и т.д. К таким библиотекам нужно относиться очень осторожно, поскольку в целях повышения эффективности соответствующие функции могут быть машинно-зависимыми и по этой причине обладать специфическими интерфейсами (хотя, скорее всего, не зависят от особенностей операционной системы). Сама по себе машинная зависимость библиотечной функции не представляет опасности, поскольку при переносе программы на компьютер с другой архитектурой все равно потребуется перекомпиляция и перекомпоновка прикладной программы, но специфичность интерфейсов может причинить большие неприятности.

Наиболее безопасным решением на сегодняшний день (при программировании на языке Си) является использование библиотек, специфицированных в стандарте языка Си. Наверное, стандартных библиотек Си окажется недостаточно в случае сложных приложений, но если при указании опции "ANSI" ваша система программирования успешно производит сборку выполняемой программы, можно быть почти уверенным, что вы не будете иметь проблем при переносе программы на компьютер, на котором установлен компилятор стандартного языка Си.

Поэтому можно сформулировать правило мобильного программирования с использованием дополнительных библиотек:

Если для разрабатываемой вами прикладной системы оказывается достаточным использование библиотек, специфицированных в стандарте языка Си, ограничьтесь использованием этих библиотек.

Если стандартных библиотек оказывается недостаточно и приходится использовать функцию из некоторой дополнительной библиотеки, поддерживаемой в вашей системе, постарайтесь проверить, насколько она стандартна. Если вы не уверены в стандартности используемой функции, то лучше напишите собственную интерфейсную функцию с известным вам интерфейсом, а при переносе прикладной программы состыкуйте эту функцию (может быть, придется ее переписать) с подходящей библиотечной функцией целевой системы (однако нет гарантии, что вам удастся ее найти).

Файлы заголовков

Использование текстовых файлов заголовков (header-файлов), которые вставляются в текст программы на языке Си с помощью директивы `include` препроцессора Си, является традиционной техникой программирования на языке Си в среде ОС UNIX, обеспечивающей синтаксическую правильность использования библиотечных функций (в том числе и системных вызовов) в прикладной программе. Ранее файлы заголовков, главным образом, содержали определения типов и символических констант (символические константы - это константы, которым сопоставлены имена посредством директивы `define` препроцессора Си), используемых в интерфейсах соответствующих библиотечных функций. Корректное применение файлов заголовков позволяло программистам не заботиться о правильности типов данных, используемых при обращении к библиотечным функциям и обработке их результатов.

Однако, традиционные файлы заголовков не гарантировали того, что набор параметров вызываемой библиотечной функции соответствовал ее интерфейсу, поскольку объявление функции, содержащее ее интерфейс, в файле компиляции отсутствовало. В лучшем случае ошибки такого рода устойчиво проявлялись во время выполнения программы, хотя далеко не всегда было просто понять их природу. В худшем случае ошибка возникала при переносе программы, поскольку одноименные библиотечные функции действительно обладали разными интерфейсами в разных средах, и в исходной операционной среде ошибки в параметрах не было.

Эту проблему удалось решить (хотя и не абсолютно) за счет введения в язык Си понятия прототипа функции. Грубо говоря, прототип функции - это часть ее объявления, содержащая только интерфейс (без тела функции). Наличие прототипа любой функции допускается в любом файле компиляции, даже не обязательно содержащем вызов этой функции. Однако, если вызов функции содержится в файле компиляции, то набор параметров вызова должен точно соответствовать интерфейсу вызываемой функции, определенному в ее прототипе.

Дальнейший ход рассуждений очевиден. Для группы родственных библиотечных функций делается общий файл заголовков, содержащий необходимые определения типов данных и символических констант, а также набор прототипов этих библиотечных функций. После включения в файл компиляции такого файла заголовков на стадии компиляции будут обнаружены все синтаксические ошибки обращения к библиотечным функциям. В предыдущем параграфе мы отметили, что это решение не абсолютно. Это действительно так, поскольку в принципе никто не может заставить программиста на языке Си включать в текст программы все требуемые файлы заголовков. Однако, такова специфика мира программирования: каждый волен усложнять свою жизнь в такой степени, в которой ему или ей это нравится.

Последнее замечание относительно файлов заголовков. В последнее время они содержат большое количество операторов условной компиляции, относящихся большей частью к определению символических констант. Дело в том, что в зависимости от версии операционной системы (мы имеем в виду версии одной линии ОС UNIX, например, UNIX System V) значения констант, используемых с одним и тем же смыслом, часто меняются. Конечно, прикладная программа не должна зависеть от таких изменений. Наличие операторов условной компиляции внутри файла заголовков разрешает эту проблему.

Поэтому последнее правило этого раздела можно сформулировать следующим образом:

При программировании на языке Си с использованием библиотечных функций используйте все требуемые файлы заголовков. Это поможет быстрее найти ошибки и повысит мобильность прикладной программы.

Мобильность на уровне исходных текстов

Материал, рассмотренный нами в предыдущем разделе, относится к вопросам мобильного программирования в связи с использованием функций операционной среды. Однако, если говорить о переносимости программ между компьютерами с разной архитектурой, имея в виду использование языка Си (не слишком высокого уровня), то нужно учитывать ряд требований, которым должна удовлетворять программа.

Особенности мобильного программирования на языке Си

Особая роль языка программирования Си состоит в том, что он, с одной стороны, позволяет писать для UNIX-систем практически столь же эффективный код, что и языки ассемблера, а с другой, является основным средством переноса программ между UNIX-системами. Можно сказать, что Си является машинно-независимым языком ассемблера для UNIX-систем. Это делает его основным средством написания эффективных и переносимых программ для этого класса вычислительных систем. Стандартизация языка сначала Американским национальным институтом стандартов (ANSI), а затем и Международной организацией по стандартам (ISO) закрепила эту роль, распространив ее и на персональные компьютеры. Будем ссылаться на версию языка Си, определенную стандартом, как на язык ANSI C.

Сказанное не означает, что любая программа, написанная на ANSI C и отлаженная в одной вычислительной системе (ВС), безусловно переносима на любую другую вычислительную систему, также имеющую компилятор языка Си, отвечающий требованиям ANSI. Однако, язык ANSI C определен таким образом, чтобы можно было писать программы, подвергающиеся минимальным изменениям при их переносе на другие вычислительные системы.

Программа на ANSI C переносима из исходной ВС в целевую, если она успешно компилируется в целевой ВС и ее работа функционально эквивалентна работе в исходной ВС.

На переносимость программы влияют особенности как аппаратного, так и программного окружения языка в исходной и в целевой ВС. Можно выделить четыре фактора, влияющих на переносимость программы:

- архитектура вычислительных систем;
- метрические ограничения компиляторов;
- алгоритмы работы компиляторов;
- особенности операционных систем.

Архитектура существенно влияет на семантику языка, а, следовательно, и на переносимость программных файлов. Во-первых, архитектура определяет множества значений арифметических типов, фиксируя тем самым семантику большинства операций языка. Во-вторых, от архитектуры, а именно, от системы команд, зависит интерпретация операций языка, остающихся недоопределенными даже после фиксирования множеств значений соответствующих типов. В-третьих, от архитектуры зависит схема размещения данных тех или иных типов в соответствующих элементах памяти.

Даже если программа удовлетворяет всем ограничениям ANSI C и прошла стадию компиляции в исходной ВС, может случиться, что в целевой ВС она эту стадию не пройдет из-за того, что некоторые метрические характеристики программы не удовлетворяют ограничениям, принятым в целевой ВС. Примерами таких характеристик являются: число уровней вложенностей составных операторов, операторов цикла и операторов выбора варианта; число описателей указателя, массива и функции, модифицирующих базовый тип в описании объекта; число выражений, вложенных друг в друга по круглым скобкам и т.п.

От алгоритмов работы компилятора зависит, например, порядок вычисления выражений, что влияет как на значения выражений, так и на вырабатываемый ими побочный эффект.

Наконец, семантика многих стандартных библиотечных функций (например, функций ввода/вывода) зависит от особенностей операционной системы.

Все перечисленные факторы учтены в определении ANSI C путем фиксирования неуточняемого (стандартом) поведения программ, неопределенного поведения программ и поведения программ, определяемого реализацией.

Неуточняемое поведение (unspecified behavior) - это поведение правильных программ с корректными данными в ситуациях, для которых стандарт не выдвигает никаких требований.

Неопределенное поведение (undefined behavior) - поведение (динамически) ошибочных программ с возможно некорректными данными или объектами с неопределенными значениями, для которых стандарт не выдвигает никаких требований. Диапазон неопределенного поведения может быть очень разнообразен: от полного игнорирования ситуации с непредсказуемыми результатами до поведения (во время трансляции или выполнения) в соответствии с документацией, описывающей характеристики среды (с выдачей диагностических сообщений или без таковой); возможны случаи преждевременного завершения трансляции или вычислений (с обязательной выдачей диагностического сообщения).

Поведение, определяемое реализацией (implementation-defined behavior) - поведение правильно написанной программы с правильными данными, которое зависит от характеристик реализации и которое должно быть документировано каждой реализацией.

В качестве общей рекомендации по написанию переносимых программ можно посоветовать, во-первых, безусловно избегать использования в программах языковых конструкций с неопределенным поведением, во-вторых, избегать конструкций с неуточняемым поведением в случаях, когда результат ее работы не является однозначным, и, наконец, минимизировать число конструкций, чье поведение определяется реализацией и существенно влияет на результат работы программы.

Другая общая рекомендация заключается в использовании возможностей препроцессора Си для локализации непереносимых фрагментов программы. Это касается использования макроимен вместо явных констант, зависящих от реализации; использования условной трансляции для включения в окончательный текст программы того или иного фрагмента в зависимости от вычислительной системы (особенно это касается конструкций, чье поведение определяется реализацией и существенно влияет на результат работы программы) и т.д.

Далее мы перечисляем все случаи неуточняемого, неопределенного и зависящего от реализации поведения программ, а, кроме того, в наименее очевидных случаях объясняем их влияние на переносимость. После этого приводятся требования стандарта к метрическим ограничениям компиляторов.

Неуточняемое поведение

Не уточняются следующие вопросы:

- Метод и время инициации статических данных.

В зависимости от того, вычисляются ли инициирующие константные выражения в окружении трансляции или в окружении выполнения программы, статические данные могут получать различные начальные значения.

- Ситуация, когда выдается печатный символ, а активная позиция находится в конце строки.
- Ситуация, когда выдается символ "шаг назад", а активная позиция находится в начале строки.
- Ситуация, когда выдается символ "горизонтальная табуляция", а активная позиция находится "на" или "за" последней определенной позицией горизонтальной табуляции.
- Ситуация, когда выдается символ "вертикальная табуляция", а активная позиция находится "на" или "за" последней определенной позицией вертикальной табуляции.

Предыдущие четыре ситуации влияют на вывод текста на дисплей.

- Представление плавающих типов.

Переносимая программа не должна использовать информацию о представлении (т.е. о битовой структуре) плавающих типов, поскольку именно в реализации плавающей арифметики существенно различаются разные вычислительные системы.

- Порядок вычисления выражений - в любом порядке, учитывающем только правила предшествования операций и расстановку скобок.
- Порядок, в котором возникают побочные эффекты.
- Порядок, в котором вычисляются параметры вызова функции и само значение этой функции.

За исключением тех случаев, когда порядок вычисления выражения зафиксирован синтаксическими правилами или указан в стандарте каким-либо другим образом (для операции вызова функции (), операций логического умножения, логического сложения, условной операции и операции перечисления выражений), порядок вычисления подвыражений и порядок возникновения побочных эффектов не уточняется. Выражение, содержащее более, чем одно вхождение одной и той же коммутативной и ассоциативной бинарной операции (*, +, &, ^, |), может свободно перегруппировываться, независимо от наличия скобок, при условии, что типы операндов или результаты от такой перегруппировки не изменятся. В переносимой программе следует избегать выражений, порядок вычисления которых существенно влияет на их значения или вырабатываемые побочные эффекты. Если же такое выражение возникает, то содержащий его оператор

всегда можно разбить на эквивалентную последовательность из нескольких операторов, не содержащих подобных выражений. Например, оператор

```
x=f()+g();
```

можно заменить на последовательность операторов

```
y=f();

x=y+g();
```

или

```
y=g();

x=f()+y;
```

в зависимости от нужного порядка вызова функций `f()` и `g()`.

Чтобы зафиксировать некоторое конкретное группирование операций, нужно присвоить значение выражения, которое требуется явно выделить, некоторому объекту данных, либо поставить перед группирующими скобками унарный оператор плюс.

- Отведение памяти под формальные параметры.

Переносимая программа не должна использовать информацию о распределении памяти под формальные параметры, поскольку не только разные компиляторы по-разному решают эту задачу, но даже один компилятор может различным образом отводить память под формальные параметры при различных режимах своей работы.

- Значение индикатора позиции файла после успешного выполнения функции `ungetc` для текстового потока до тех пор, пока не будут введены или уничтожены все запомненные символы.
- Подробности о значении, запоминаемом в случае успешной работы функции `fgetpos`.
- Подробности о значении, вырабатываемом для текстового потока в случае успешной работы функции `ftell`.
- Порядок и взаимное расположение областей памяти, захватываемых функциями `calloc`, `malloc` и `realloc`.
- Какой из двух элементов, оказавшихся равными при сравнении, возвращается функцией `bsearch`.
- Порядок расстановки в отсортированном функцией `qsort` массиве двух элементов, оказавшихся равными при сравнении.
- Структура календарного времени, возвращаемого функцией `time`.

Переносимая программа не использует перечисленную информацию, поскольку она либо различается для разных реализаций языка, либо даже является случайной в рамках одной реализации.

Неопределенное поведение

Поведение не определяется для следующих ситуаций:

- В исходной программе обнаружен символ, не входящий в требуемый набор. Исключение делается для препроцессорных лексем, символьных и строковых констант, а также примечаний.
- Делается попытка модифицировать строковую константу.
- Идентификаторы, которые должны обозначать одну и ту же сущность, различаются хотя бы одним символом.
- В символьной или строковой константе обнаружена неизвестная управляющая последовательность.
- Лексически первое описание функции или объекта данных с внешней связью не имеет файловой области видимости, а последующее описание лексически идентичного идентификатора имеет либо внутреннюю, либо внешнюю связь, что противоречит первому описанию.
- Арифметическое преобразование дает результат, который не может быть представлен в отведенном пространстве.
- Арифметическая операция неверна (например, деление на 0) или выдает результат, который нельзя представить в отведенном пространстве (например, переполнение или потеря значимости).
- Число фактических параметров вызова не согласуется с числом формальных параметров функции, которая не имеет действующего в данной области видимости прототипа.
- Типы фактических параметров вызова после расширения не согласуются с расширенными типами формальных параметров функции, которая не имеет действующего в данной области видимости прототипа и не имеет прототипа, действующего в области видимости, соответствующей области определения функции.
- Прототип функции имеется в области видимости, соответствующей области определения функции, формальный параметр описан с типом, который изменяется в результате действия расширений типа, проводимых по умолчанию, а функция вызывается, когда в области видимости нет семантически эквивалентного прототипа.
- Вызывается функция, обрабатывающая переменное число параметров, но прототип с эллиптической нотацией отсутствует в данной области видимости.
- Вызывается функция с прототипом, видимым в данной области, ее формальный параметр описан с типом, который изменяется в результате действия расширений типа, проводимых по умолчанию, но в области определения функции не видно семантически эквивалентного прототипа функции.
- Встретилась неверная ссылка на массив, ссылка на пустой указатель или ссылка на объект, размещенный в области автоматически распределяемой памяти завершившегося блока.
- Указатель на функцию преобразуется в указатель на функцию другого типа и используется для вызова функции, тип которой отличается от первоначального.
- Указатель на объект, не являющийся элементом массива, используется в операции прибавления или вычитания константы.
- Вычисляется разность указателей, относящихся к разным массивам.
- Результат выражения сдвигается на отрицательную величину или на величину, большую или равную (в битах) размеру сдвигаемого результата.
- Сравниваются указатели, относящиеся к разным составным объектам.
- Значение объекта присваивается перекрывающемуся по памяти объекту.
- Делается попытка изменить объект, описанный как константа, с помощью указателя на тип, в котором нет атрибута `const`.
- Объект, описанный с атрибутом `volatile`, указывается с помощью указателя на тип, не имеющего такого же атрибута.

- Описания объекта, имеющего внешнюю связь, в двух разных файлах или в разных областях видимости одного файла, дают этому объекту разные типы.
- Значение автоматического неинициированного объекта используется до первого присваивания.
- Используется результат работы функции, которая, однако, не возвращает никакого значения.
- Функция, обрабатывающая переменное число параметров, определяется без списка типов параметров в эллиптической нотации.
- Фактический параметр макровызова не имеет ни одной препроцессорной лексемы.
- Внутри списка параметров макровызова имеются препроцессорные лексемы, которые могут быть проинтерпретированы как директивы препроцессора.
- В результате выполнения препроцессорной операции слияния лексем (##) получается неверная препроцессорная лексема.
- Эффект, возникающий в программе при переопределении зарезервированного внешнего идентификатора.
- Параметр `identifier` в макровызове `offset` соответствует битовому полю записи.
- Фактический параметр библиотечной функции имеет неверное значение, если только поведение этой функции в подобном случае не описано явно.
- Библиотечная функция, обрабатывающая переменное число параметров, не описана.
- Для доступа к настоящей функции `assert` использована макродирективa #`undef`.
- Фактический параметр функции, обрабатывающей символы, выходит за область определения.
- Вызов функции `setjmp` производится в ином контексте, нежели при сравнении с целочисленным выражением из констант в переключателе или в условном операторе.
- Значение автоматического объекта, не имеющего атрибута `volatile`, изменилось между вызовами `setjmp` и `longjmp`.
- Функция `longjmp` вызывается из динамически вложенной программы обработки сигнала.
- Сигнал возникает не в результате работы функций `abort` или `raise`, а при обработке сигнала вызывается библиотечная функция, не являющаяся самой функцией `signal`, или со статическим объектом проделывается не присваивание ему значения статической переменной с атрибутом `volatile` типа `sig_atomic_t`.
- Параметр `parmN` макроопределения `va_start` описывается в классе регистровой памяти.
- При вызове макроимени `va_arg` очередного фактического параметра не оказалось.
- Тип фактического параметра из списка параметров не согласуется с типом, указанным в макровызове `va_arg`.
- Функция `va_end` вызывается без предварительного обращения к макровызову `va_start`.
- Из функции с переменным числом параметров, список которых был проинициирован с помощью макровызова `va_start`, возврат производится до вызова `va_end`.
- Формат в функциях `fprintf` и `fscanf` не соответствует списку фактических параметров.
- В формате функций `fprintf` или `fscanf` обнаружена неверная спецификация преобразования.
- Среди спецификаторов преобразования для спецификации, не входящей в список o, x, X, e, E, f, g и G встретился признак #.

- Фактическим параметром функции `fprintf`, не соответствующим преобразованиям %s и %p, является составной объект или указатель на составной объект.
- Отдельное преобразование в функции `fprintf` породило более 509 выходных символов.
- Фактическим параметром преобразования %p функции `fscanf` является значение указателя, выданное при преобразовании %p функцией `fprintf` во время предыдущих запусков программы.
- Результат преобразования, выполняемого функцией `fscanf`, не может быть представлен в объеме памяти, отведенной для него, или полученный объект имеет неподходящий тип.
- Результат преобразования строки в число с помощью функций `atof`, `atoi` или `atol` не может быть представлен.
- Фактический параметр функций `free` или `realloc` не совпадает с ранее полученными указателями, выработанными функциями `calloc`, `malloc` или `realloc`, или указывается объект, ранее уничтоженный вызовом функций `free` или `realloc`.
- Ссылка на память, освобожденную функциями `free` или `realloc`.
- При вызове из функции `exit` функция, зарегистрированная обращением к `atexit`, производит доступ к автоматическому объекту программы.
- Результат целочисленных арифметических функций (`abs`, `div`, `labs` или `ldiv`) не может быть представлен.
- Массив, в который идет запись копированием или конкатенацией, слишком мал.
- Функции `memcpy`, `strcpy` или `strncpy` копируют объект в перекрывающийся с ним по памяти другой объект.
- В формате функции `strftime` обнаружена неверная спецификация преобразования.

Все перечисленные ситуации являются ошибочными, однако разные реализации могут по-разному реагировать на них. Может даже случиться, что в некоторых реализациях программы с неопределенным поведением работают и выдают нужные результаты. Однако такие программы, как правило, невозможно перенести на другую вычислительную систему.

Например, используя в расчетной программе неверные арифметические операции (деление на ноль или операции, приводящие к переполнению или потере значимости), можно добиться удовлетворительной, с точки зрения конечного результата, работы этой программы за счет использования нюансов обработки таких исключительных ситуаций в рамках конкретной вычислительной системы. На других же вычислительных системах эта программа либо вообще не будет работать, либо будет выдавать неудовлетворительные результаты. Больше того, может потребоваться даже изменение алгоритма, реализуемого программой, из-за невозможности воспроизвести использованные нюансы исходной вычислительной системы хотя бы потому, что программист мог и не знать обо всех использованных тонкостях аппаратуры по принципу "есть результат и ладно" (кстати, техническая документация может и не содержать описания всех тонкостей).

Возникновения ситуаций с неопределенным поведением можно, а при разработке переносимых программ, безусловно, нужно избегать.

Поведение, зависящее от реализации

Каждая реализация должна описать поведение во всех ситуациях, перечисленных в этом разделе.

Семантика фактических параметров функции `main`.

Для облегчения переноса программы полезно локализовать обработку внешних аргументов.

Число значащих начальных символов (сверх 31) в идентификаторе без внешней связи.

В переносимой программе не используется свыше 31 значащего символа в идентификаторах без внешней связи.

Число значащих начальных символов (сверх 6) в идентификаторе с внешней связью.

В переносимой программе не используется свыше 6 значащих символов в идентификаторах с внешней связью.

Имеет ли значение регистр символов, входящих в идентификаторы с внешней связью.

При разработке переносимых программ лучше исходить из того, что регистр символов, входящих в идентификатор с внешней связью, не имеет значения (т.е. не различаются заглавные и прописные буквы).

Символы входного алфавита, кроме явно определенных в стандарте.

Это касается, в основном, символов, используемых в символьных и строковых константах (например, русские буквы).

Символы из набора времени выполнения (за исключением пустого символа и (в окружении выполнения) явно определенных символов входного символьного набора) и их коды.

В переносимых программах нежелательно использование информации о кодах символов, поскольку они могут различаться в разных реализациях.

Соответствие символов входного алфавита (в символьных и строковых константах) символам алфавита времени выполнения.

В основном это касается управляющих символов. Например, символ "конец строки" (\n) в разных реализациях может быть представлен в потоках ввода-вывода различными последовательностями кодов. Надо стараться писать программу так, чтобы ее поведение не зависело от конкретного представления управляющих символов в окружении выполнения.

Число и порядок символов в целом.

Эти различия несущественны в самостоятельных программах, которые не позволяют себе играть типами (например, преобразуя указатель на целое в указатель на символы и проверяя содержимое памяти по указателю), но могут проявиться при обработке данных, поступающих извне.

Число и порядок следования разрядов в символах из набора символов времени выполнения.

Значение символьной константы, состоящей из символа или управляющей последовательности, не представимой в алфавите времени выполнения.

Переносимой программе не следует использовать информацию этих двух пунктов.

Значение символьной константы, состоящей более, чем из одного символа.

В переносимой программе не следует использовать символьные константы более, чем из одного символа.

Следует ли трактовать "простые" символьные объекты как знаковые или беззнаковые.

Переносимая программа не должна зависеть от того, является ли тип char знаковым или беззнаковым.

Представление и наборы значений различных целочисленных типов.

В переносимой программе лучше всего исходить из минимальных наборов значений, зафиксированных стандартом, а также из той минимальной информации о представлении, которая в приводится в стандарте.

Результат преобразования целого к более короткому знаковому целому или результат преобразования беззнакового целого к знаковому целому той же длины, если значение не может быть представлено.

Переносимая программа не использует эту информацию.

Результаты поразрядных операций над знаковыми целыми.

В переносимой программе следует использовать только такие поразрядные операции, результат которых не зависит от реализации.

Знак остатка целочисленного деления.

Переносимая программа не использует эту информацию.

Является ли сдвиг вправо значения знакового целочисленного типа логическим или арифметическим.

Переносимая программа не должна зависеть от вида сдвига вправо знаковых целых.

Представление и наборы значений различных типов вещественных чисел.

Переносимая программа не зависит от представления вещественных чисел. Наборы значений вещественных типов влияют на точность вычислений.

Способ округления, когда вещественное число преобразуется к более узкому вещественному числу.

В переносимой программе лучше всего исходить из того, что способ округления неизвестен.

Тип целого, которое может вместить максимальный размер массива, то есть тип `size_t` - тип результата операции `sizeof`.

Результат преобразования указателя в целое и наоборот.

Тип целого, которое может вместить разность между двумя указателями на один и тот же массив - `ptrdiff_t`.

Переносимая программа не должна использовать информацию предыдущих трех пунктов.

Элемент смеси `union` используется как элемент другого типа.

Переносимая программа не должна осуществлять доступ к элементу смеси после того, как был изменен элемент смеси другого типа, поскольку в этом случае используется информация о битовой структуре представления значения соответствующего типа.

Дополнение пустот и выравнивание элементов записей.

Это обычно не доставляет проблем, если только двоичные данные, записанные одной реализацией, не читаются другой. Конечно же, не следует использовать эту информацию в переносимой программе.

Считается ли "простое" целое битовое поле знаковым или беззнаковым.

Переходит ли битовое поле, не умещающееся в одном целом, в следующее.

Порядок расположения битовых полей в целом.

Может ли битовое поле пересекать физические границы ячеек памяти.

Переносимая программа не должна использовать всю эту информацию.

Максимальное число описателей, которые могут модифицировать базовый тип.

Переносимой программе нужно исходить из того, что любая реализация должна допускать использование в модификации базового типа, либо непосредственно, либо через эквивалентность типов, по крайней мере 12 описателей указателей, массивов и функций (в любых комбинациях).

Максимальное число вариантов в переключателе.

Переносимая программа должна исходить из того, что число вариантов в переключателе не должно превышать 255.

Будет ли значение односимвольной символьной константы в выражении, управляющем условным включением фрагментов программ, совпадать со значением такой же константы в наборе символов окружения выполнения. Может ли такая константа иметь отрицательное значение.

Метод связи с входными файлами, подлежащими включению в программу.

Обработка имен в кавычках, относящихся к включаемым файлам.

Поведение каждой директивы `#pragma`.

Определение имен __DATE__ и __TIME__, когда, соответственно дата и время трансляции не может быть доступно.

Константа, получающаяся при подстановке макроопределения NULL, обозначающая пустой указатель.

Предыдущие 6 пунктов описывают зависящее от реализации поведение препроцессора. Остальные пункты описывают определяемое реализацией поведение библиотечных программ.

Диагностическое сообщение и способ завершения программы, применяемый в функции `assert`.

Наборы символов, проверяемые в функциях `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint` и `isupper`.

Значения, выдаваемые математическими функциями при возникновении ошибок области определения.

Устанавливают ли математические функции целое выражение `errno` в положение ERANGE при возникновении потери значимости.

Набор сигналов для функции signal.

Семантика каждого сигнала, распознаваемого библиотечной функцией `signal`.

Обработка умолчаний и входов в программу для каждого вида сигналов, распознаваемых функцией `signal`.

Восстанавливается ли стандартная обработка, если при обработке сигнала функцией, указанной при вызове функции `signal`, возникает сигнал SIGILL.

Нужно ли заканчивать последнюю строку текстового потока символом "конец строки".

Появятся ли при вводе обычные пробелы, записанные в текстовый поток непосредственно перед символом конца строки текста.

Количество символов NULL, которые дописываются к двоичному потоку.

Характеристики буферизации файлов.

Существует ли файл нулевой длины.

Правила образования правильных имен файлов.

Может ли один файл открываться много раз.

Результат выполнения функции `remove` над открытым файлом.

Эффект работы функции `rename`, если файл с новым именем существовал ранее.

Выходная строка, получающаяся при работе преобразования %p в функции `fprintf`.

Входная строка, поступающая для преобразования %p в функции `fscanf`.

Интерпретация символа ^, который есть ни первый, ни последний символ в списке сканирования в преобразовании %[в функции `fscanf`.

Значение, которое получает `errno` от функций `fgetpos` и `ftell` в случае неудачи.

Сообщения, выдаваемые функцией `perror`.

Поведение функций `calloc`, `malloc` и `realloc` в случае, если размер запрошенной памяти равен нулю.

Поведение функции `abort` по отношению к открытым и временным файлам.

Статус, возвращаемый функцией `exit`, если значение фактического параметра не равно нулю, или значениям макроимен EXIT_SUCCESS и EXIT_FAILURE.

Набор имен окружения и метод изменения списка окружения, используемый функцией `getenv`.

Содержание и режим выполнения командной строки функцией `system`.

Знак значения, возвращаемого функцией сравнения (`memcmp`, `strcmp` или `strncmp`), если первая пара различающихся символов разнится в старшем разряде.

Содержание строк сообщений об ошибках, возвращаемых функцией `strerror`.

Местный временной пояс и летнее время.

Точка отсчета для функции `clock`.

Метрические ограничения переносимой программы

Переносимая программа должна удовлетворять следующим метрическим ограничениям:

- 15 уровней вложенности составных операторов, операторов цикла и операторов выбора варианта.
- 6 уровней вложенности условной трансляции.
- 12 описателей указателя, массива и функции, модифицирующих базовый тип в описании объекта.
- 127 выражений, вложенных друг в друга по круглым скобкам.
- 31 значащий символ в начале идентификатора с внутренней связью или имени макроопределения.
- 6 значащих символов в начале имен, имеющих внешнюю связь.
- 511 внешних имен в одном исходном файле.
- 127 имен в одном блоке.
- 1024 имени макроопределений, одновременно действующих в одном исходном файле.
- 31 параметр в вызове или определении функции.
- 31 параметр в макровызове или макроопределении с параметрами.

- 509 символов в одной логической исходной строке.
- 509 символов в строковой константе (после конкатенации).
- 32767 байтов для размещения объекта.
- 8 уровней вложенности по включаемым файлам.
- 255 меток выбора варианта в переключателях.

Обеспечение независимости от особенностей версии ОС UNIX

Специфичным видом мобильности приложений на уровне исходных текстов является возможность их выполнения с несколькими версиями одного и того же варианта ОС UNIX, включая ранние версии, далекие от современных стандартов. Достаточно часто поздние версии не обеспечивают полной совместимости с более ранними версиями, поскольку такая совместимость не дала бы возможности добиться в поздних версиях соответствия стандартам.

По-видимому, единственным на текущий момент практическим приемом для достижения такого рода мобильности приложений на уровне исходных текстов является развитое применение операторов условной компиляции в текстах программ (условной компиляции на уровне файлов включения часто оказывается недостаточно, поскольку в зависимости от версии системы в прикладной системе приходится использовать разные комбинации системных вызовов и других библиотечных функций). Обычно в основных файлах включения поддерживаются символические константы, значения которых позволяют судить об особенностях используемой версии ОС. Опираясь на значения этих констант, можно добиться того, что правильно написанная прикладная программа будет правильно компилироваться (и собираться) в среде конкретной версии операционной системы. Наличие единого текста облегчает сопровождение прикладной программы и облегчает достижение его одинаковой функциональности при работе с разными версиями системы (хотя такие тексты, переполненные операторами условной компиляции, обычно очень трудно читаются; можно только рекомендовать по мере возможности локализовать куски программы, зависящие от версии ОС).

Бинарная совместимость

Если обычно достижение мобильности прикладных программ является целью прикладных программистов, то иногда достижение бинарной совместимости при выполнении прикладных программ является задачей разработчиков операционных систем. Под бинарной совместимостью операционной системы О2 с операционной системой О1 понимается возможность выполнения в среде О2 без перекомпиляции (а, возможно, и без перекомпоновки) приложений, написанных для выполнения в среде О1. Естественно, что бинарная совместимость двух операционных сред теоретически достижима только в том случае, когда обе операционные системы О1 и О2 базируются на некоторой общей аппаратной платформе (реально, чаще всего приходится слышать о бинарной совместимости разных вариантов ОС UNIX, работающих на платформах Intel).

Двоичная совместимость новой операционной системы с некоторой существующей ОС требуется в том случае, когда, во-первых, необходимо доказать пользователям, что новая система не только обладает новыми качествами, но и настолько технологична, что может выполнять существующие приложения даже без потребности их перекомпиляции. Во-вторых, двоичная совместимость позволяет немедленно сделать доступным в новой операционной среде весь накопленный в старой среде багаж приложений (исходные тексты которых будут, скорее всего, недоступны), что может оказаться очень

существенным для потенциальных конечных пользователей (потребителей приложений) новой системы.

Возможности достижения бинарной совместимости

Конечно, прежде всего требуется полная аппаратная совместимость используемых платформ (по крайней мере, на уровне пользователя). Далее возможны два варианта. В первом, более простом варианте обеспечивается двоичная совместимость в смысле использования в новой операционной среде объектных файлов, откомпилированных в расчете на прежнюю операционную среду. Для получения выполняемой программы в новой среде требуется перекомпоновка программы (конечно, для этого компоновщик выполняемых программ новой ОС должен понимать структуру объектных модулей старой системы). Этот вариант близок к подходу переносимости программ на уровне исходных текстов, поскольку старые объектные модули содержат только пользовательский код и вызовы библиотечных функций и, очевидно, будут выполняться в новой среде без проблем. Все, что остается сделать (но это очень непростая задача) - это добиться полной совместимости со старой средой на уровне системных библиотек всех уровней. Нужно заметить, что этот вид бинарной совместимости не очень эффектен и не очень практичен, поскольку наборы объектных файлов приложений получить не намного проще, чем их исходные тексты. Обычно доступны выполняемые программы.

Во втором варианте в новой операционной системе возможно выполнение построенных в старой операционной среде выполняемых файлов. Это уже полностью скомпонованные программы, содержащие, кроме обычных пользовательских команд, только специальные команды вызова функций ядра операционной системы (обычно, разновидности команды `trap`). С одной стороны, для обеспечения этого вида бинарной совместимости не требуется воспроизводить весь набор библиотек старой операционной среды, но, с другой стороны, требуется полностью воспроизвести интерфейс с ядром старой операционной системы на самом низком уровне. Понятно, что это выполнимая, но трудная техническая задача (поскольку детали этого интерфейса обычно публично недоступны).

Преимущества и ограничения

О преимуществах подхода бинарной совместимости мы уже сказали в начале этого раздела: привлечение прикладных программистов возможностью использовать свои старые программы без каких-либо переделок и привлечение конечных пользователей возможностью использовать все накопленные приложения.

Ограничением систем, обеспечивающих бинарную совместимость является то, что позволяя использовать старые приложения, они тормозят использование новых качеств системы (а новые системы всегда обладают новыми качествами, в противном случае их было бы бессмысленно делать). Кроме того, в наше время аппаратные архитектуры развиваются так быстро, что их развитие тормозится требованиями аппаратной совместимости платформ одной линии даже на пользовательском уровне. Как кажется (это субъективное мнение автора), в этих условиях достижение бинарной совместимости операционных систем на одной аппаратной платформе становится слишком дорогостоящей и неблагодарной задачей.

Традиционные средства интерактивного интерфейса пользователей

По своей исходной задумке ОС UNIX является типичной интерактивной операционной системой, или системой с разделением времени. Это означает, что каждый пользователь системы взаимодействует с системой со своего собственного терминала в интерактивном режиме, задавая системе команды и получая на экран своего терминала ответные сообщения системы. В общем, эта картина остается истинной для всех современных вариантов ОС UNIX. Но в существенных деталях она сильно отличается от того, что было несколько лет тому назад. Основное отличие заключается в способе организации интерактивного интерфейса с системой.

Когда создавались первые варианты ОС UNIX, единственным практически доступным (и сравнительно удобным) аппаратным средством интерактивного взаимодействия с вычислительной системой являлись алфавитно-цифровые терминалы, способные вводить и выводить строки символов. Поэтому исторически базовым средством взаимодействия системы с пользователем является строчный интерфейс: пользователь вводит со своего терминала некоторую строку символов, и если система понимает смысл этой строки, то она выполняет соответствующие действия и выдает на экран пользователя соответствующие результаты. (Конечно, здесь смысл терминов "строка символов", "система", "понимает" и т.д. нуждается в уточнении, что и будет сделано ниже.)

Разные алфавитно-цифровые терминалы обладали разными возможностями. Например, некоторые из них обеспечивали режимы реверсного вывода, полутонов, псевдографики и т.д. Однако система должна была работать с любым из этих терминалов. Поэтому использовались минимальные возможности. Фактически, алфавитно-цифровой терминал в базовом интерфейсе с ОС UNIX используется в режиме электронного телетайпа. В этом разделе мы рассмотрим интерфейсные возможности ОС UNIX этого уровня.

Широкое распространение дешевых цветных терминалов с развитыми графическими возможностями (и, в особенности, повсеместное использование персональных компьютеров с графическими интерфейсами) стимулировало переход ОС UNIX на использование графических интерфейсов взаимодействия с пользователем. В настоящее время невозможно найти современный вариант ОС UNIX, в котором не поддерживалась бы возможность использования графических терминалов в многооконном режиме с применением соответствующего графического интерфейса в каждом окне. Соответствующие средства мы рассмотрим в следующем разделе.

Однако сразу заметим, что нам не приходилось еще видеть графического терминала, подключенного к UNIX-системе, хотя бы одно окно которого не использовалось бы в качестве аналога традиционного алфавитно-цифрового терминала для взаимодействия с системой в традиционном режиме (это не обязательно, но экономит время). Более того, многие профессиональные программисты предпочитают использовать традиционные интерфейсы, осознавая, насколько большие возможности они обеспечивают и насколько меньшие порождают накладные расходы. Поэтому (по крайней мере, на сегодняшний день) без знания основ традиционного интерфейса с ОС UNIX обойтись все еще нельзя (если, конечно, хотеть использовать систему профессионально).

Командные языки и командные интерпретаторы

Как и в большинстве интерактивных систем, традиционный интерфейс с пользователем ОС UNIX основан на использовании командных языков. Выражаясь несколько

тавтологично, можно сказать, что командный язык - это язык, на котором пользователь взаимодействует с системой в интерактивном режиме. Такой язык называется командным, поскольку каждую строку, вводимую с терминала и отправляемую системе, можно рассматривать как команду пользователя по отношению к системе. Одним из достижений ОС UNIX является то, что командные языки этой операционной системы являются хорошо определенными (не очень хороший русский термин, соответствующий совершенно однозначному английскому термину well-defined) и содержат много средств, приближающих их к языкам программирования.

Если рассматривать категорию командных языков с точки зрения общего направления языков взаимодействия человека с компьютером, то они, естественно, относятся к семейству интерпретируемых языков. Коротко охарактеризуем разницу между компилируемыми и интерпретируемыми компьютерными языками. Язык называется компилируемым, если требует, чтобы любая законченная конструкция языка была настолько замкнутой, чтобы обеспечивала возможность изолированной обработки без потребности привлечения дополнительных языковых конструкций. В противном случае понимание языковой конструкции не гарантируется. Житейским примером компилируемого языка является литературный русский язык. Ни один литературный редактор не примет от вас незаконченное сочинение, в котором имеются ссылки на еще не написанные части. Процесс компиляции (литературного редактирования в нашем примере) требует замкнутости языковых конструкций.

Основным преимуществом интерпретируемых языков является то, что в случае их использования программа пишется "инкрементально" (в пошаговом режиме), т.е. человек принимает решение о своем следующем шаге в зависимости от реакции системы на предыдущий шаг. В принципе, предпочтение компилируемых или интерпретируемых языков является предметом личного вкуса конкретного индивидуума (нам известны крупные авторитеты в области программирования - например, Д.Б. Подшивалов,- которые абсолютно уверены, что любая хорошая программа должна быть сначала написана на бумаге и отлажена за столом).

Особенностью командных языков является то, что в большинстве случаев они не используются для программирования в обычном смысле этого слова, хотя на развитом командном языке можно написать любую программу. По нашему мнению, правильным стилем использования командного языка является его применение в основном для непосредственного взаимодействия с системой с привлечением возможностей составления командных файлов (скриптов или сценариев в терминологии ОС UNIX) для экономии повторяющихся рутинных процедур.

Программы, предназначенные для обработки конструкций командных языков, называются командными интерпретаторами. В отличие от компилируемых языков программирования (таких, как Си или Паскаль), для каждого из которых обычно существует много различных компиляторов, командный язык, как правило, неразрывно связан с соответствующим интерпретатором. Когда ниже мы будем говорить о различных представителях командных языков ОС UNIX, относящихся к семейству shell, то каждый раз под одноименным названием мы будем подразумевать и соответствующий интерпретатор.

Общая характеристика командных языков

В этом пункте и далее в данном разделе мы будем более конкретно говорить о командных языках семейства `shell`. Основное назначение этих языков (их разновидностей

существует достаточно много, но мы рассмотрим только три наиболее распространенные варианта - Bourne-shell, C-shell и Korn-shell) состоит в том, чтобы предоставить пользователям удобные средства взаимодействия с системой. Что это означает? Языки не даром называются командными. Они предназначены для того, чтобы дать пользователю возможность выполнять команды, предназначенные для исполнения некоторых действий операционной системы. Существует два вида команд.

Собственные команды `shell` (такие как `cd`, `echo`, `exec` и т.д.) выполняются непосредственно интерпретатором, т.е. их семантика встроена в соответствующий язык. Имена других команд на самом деле являются именами файлов, содержащих выполняемые программы. В случае вызова такой команды интерпретатор командного языка с использованием соответствующих системных вызовов запускает параллельный процесс, в котором выполняется нужная программа. Конечно, смысл действия подобных команд является достаточно условным, поскольку зависит от конкретного наполнения внешних файлов. Тем не менее, в описании каждого языка содержатся и характеристики "внешних команд" (например, `find`, `grep`, `cc` и т.д.) в расчете на то, что здравомыслящие пользователи (и их администраторы) не будут изменять содержимое соответствующих файлов.

Существенным компонентом командного языка являются средства, позволяющие разнообразными способами комбинировать простые команды, образуя на их основе составные команды. В семействе языков `shell` возможны следующие средства комбинирования. В одной командной строке (важный термин, означающий единицу информационного взаимодействия с командным интерпретатором) можно указать список команд, которые должны выполняться последовательно, или список команд, которые должны выполняться "параллельно" (т.е. независимо одна от другой).

Очень важной особенностью семейства языков `shell` являются возможности перенаправления ввода/вывода и организации конвейеров команд. Естественно, эти возможности опираются на базовые средства ОС UNIX (см. п. 2.1.8). Кратко напомним, в чем они состоят. Для каждого пользовательского процесса (а внешние команды `shell` выполняются в рамках отдельных пользовательских процессов) предопределены три выделенных дескриптора файлов: файла стандартного ввода (standard input), файла стандартного вывода (standard output) и файла стандартного вывода сообщений об ошибках (standard error). Хорошим стилем программирования в среде ОС UNIX является такой, при котором любая программа читает свои вводные данные из файла стандартного ввода, а выводит свои результаты и сообщения об ошибках в файлы стандартного вывода и стандартного вывода сообщений об ошибках соответственно. Поскольку любой порожденный процесс "наследует" все открытые файлы своего предка (см. п. 2.1.7), то при программировании команды рекомендуется не задумываться об источнике вводной информации программы, а также конкретном ресурсе, поддерживающим вывод основных сообщений и сообщений об ошибках. Нужно просто пользоваться стандартными файлами, за конкретное определение которых отвечает процесс-предок (заметим, что по умолчанию все три файла соответствуют вводу и выводу на тот терминал, с которого работает пользователь).

Что обеспечивает такая дисциплина работы? Прежде всего возможность создания программ, "нейтральных" по отношению к источнику своих вводных данных и назначению своих выводных данных. Собственно, на этом и основаны принципы перенаправления ввода/вывода и организации конвейера команд. Все очень просто. Если вы пишете в командной строке конструкцию

```
com1 par1, par2, ..., parn > file_name ,
```

то это означает, что для стандартного вывода команды `com1` будет использоваться файл с именем `file_name`. Если вы пишете

```
file_name < com1 par1, par2, ..., parn ,
```

то команда `com1` будет использовать файл с именем `file_name` в качестве источника своего стандартного ввода. Если же вы пишете

```
com1 par1, par2, ..., parn | com2 par1, par2, ..., parm ,
```

то в качестве стандартного ввода команды `com2` будет использоваться стандартный вывод команды `com1`. (Конечно, при организации такого рода "конвейеров" используются программные каналы, см. п. 3.4.4.)

Конвейер представляет собой простое, но исключительно мощное средство языков семейства `shell`, поскольку позволяет во время работы динамически создавать "комбинированные" команды. Например, указание в одной командной строке последовательности связанных конвейером команд

```
ls -l | sort -r
```

приведет к тому, что подробное содержимое текущего каталога будет отсортировано по именам файлов в обратном порядке и выдано на экран терминала. Если бы не было возможности комбинирования команд, до для достижения такой возможности потребовалось бы внедрение в программу `ls` возможностей сортировки.

Последнее, что нам следует обсудить в этом пункте, это существо команд семейства языков `shell`. Различаются три вида команд. Первая разновидность состоит из команд, встроенных в командный интерпретатор, т.е. составляющих часть его программного кода. Эти команды предопределены в командном языке, и их невозможно изменить без переделки интерпретатора. Команды второго вида - это выполняемые программы ОС UNIX. Обычно они пишутся на языке Си по определенным правилам (см. п. 5.2.1). Такие команды включают стандартные утилиты ОС UNIX, состав которых может быть расширен любым пользователем (если, конечно, он еще способен программировать). Наконец, команды третьего вида (так называемые скрипты языка `shell`) пишутся на самом языке `shell`. Это то, что традиционно называлось командным файлом, поскольку на самом деле представляет собой отдельный файл, содержащий последовательность строк в синтаксисе командного языка.

Базовые возможности семейства командных интерпретаторов

Здесь под командным интерпретатором мы будем понимать не соответствующую программу, а отдельный сеанс ее выполнения. Если в случае компилируемых программ следует различать наличие готовой к выполнению программы и факт запуска такой программы, то в случае интерпретируемых программ приходится различать случаи наличия интерпретатора, программы, которая может быть проинтерпретирована, и запуска интерпретатора с предоставлением ему интерпретируемой программы. Основным свойством динамически выполняемой программы (неважно, является ли она предварительно откомпилированной или интерпретируемой) является наличие ее

динамического контекста, в частности, набора определенных переменных, содержащих установленные в них значения.

В случае командных интерпретаторов семейства `shell` имеется два вида динамических контекстов выполнения интерпретируемой программы. Первый вид контекста состоит из предопределенного набора переменных, которые существуют (и обладают значениями) независимо от того, какая программа интерпретируется. В терминологии ОС UNIX этот набор переменных называется окружением или средой сеанса выполнения командной программы. С другой стороны, программа при своем выполнении может определить и установить дополнительные переменные, которые после этого используются точно так же, как и предопределенные переменные. Спецификой командных интерпретаторов семейства shell (связанной с их ориентацией на обеспечение интерфейса с операционной системой) является то, что все переменные shell-программы (сеанса выполнения командного интерпретатора) являются текстовыми, т.е. содержат строки символов.

Любая разновидность языка `shell` представляет собой развитый компьютерный язык, и объем этого курса не позволяет представить в деталях хотя бы один из них. Однако в следующих подразделах мы постараемся кратко познакомить вас с особенностями трех распространенных вариантов языка shell.

Bourne-shell

Bourne-shell является наиболее распространенным командным языком (и одновременно командным интерпретатором) системы UNIX. Вот основные определения языка Bourne-shell (конечно, мы приводим неформальные определения, хотя язык обладает вполне формализованным синтаксисом):

- Пробел - это либо символ пробела, либо символ горизонтальной табуляции.
- Имя - это последовательность букв, цифр или символов подчеркивания, начинающаяся с буквы или подчеркивания.
- Параметр - имя, цифра или один из символов `*, @, #, ?, –, $, !`.
- Простая команда - последовательность слов, разделенных пробелами. Первое слово простой команды - это ее имя, остальные слова - аргументы команды (имя команды считается ее нулевым аргументом - см. п. 5.2.1).
- Метасимволы. Аргументы команды (которые обычно задают имена файлов) могут содержать специальные символы (метасимволы) "*", "?", а также заключенные в квадратные скобки списки или указанные диапазоны символов. В этом случае заданное текстовое представление параметра называется шаблоном. Указание звездочки означает, что вместо указанного шаблона может использоваться любое имя, в котором звездочка заменена на произвольную текстовую строку. Задание в шаблоне вопросительного знака означает, что в соответствующей позиции может использоваться любой допустимый символ. Наконец, при использовании в шаблоне квадратных скобок для генерации имени используются все указанные в квадратных скобках символы. Команда применяется в цикле для всех осмысленных сгенерированных имен.
- Значение простой команды - это код ее завершения, если команда заканчивается нормально, либо 128 + код ошибки, если завершение команды не нормальное (все значения выдаются в текстовом виде).
- Команда - это либо простая команда, либо одна из управляющих конструкций (специальных встроенных в язык конструкций, предназначенных для организации сложных shell-программ).
- Командная строка - текстовая строка на языке `shell`.

- shell-процедура (shell-script) - файл с программой, написанной на языке `shell`.
- Конвейер - последовательность команд, разделенных символом `"|"`. При выполнении конвейера стандартный вывод каждой команды конвейера, кроме последней, направляется на стандартный вход следующей команды. Интерпретатор `shell` ожидает завершения последней команды конвейера. Код завершения последней команды считается кодом завершения всего конвейера.
- Список - последовательность нескольких конвейеров, соединенных символами `";", "&", "&&", "||",` и, может быть, заканчивающаяся символами `";"` или `"&"`. Разделитель между конвейерами `";"` означает, что требуется последовательное выполнение конвейеров; `"&"` означает, что конвейеры могут выполняться параллельно. Использование в качестве разделителя символов `"&&"` (и `"||"`) означает, что следующий конвейер будет выполняться только в том случае, если предыдущий конвейер завершился с кодом завершения `"0"` (т.е. абсолютно нормально). При организации списка символы `";"` и `"&"` имеют одинаковые приоритеты, меньшие, чем у разделителей `"&&"` и `"||"`.
- В любой точке программы может быть объявлена (и установлена) переменная с помощью конструкции `"имя = значение"` (все значения переменных - текстовые). Использование конструкций `$имя` или `${имя}` приводит к подстановке текущего значения переменной в соответствующее слово.
- Предопределенными переменными Bourne-shell, среди прочих, являются следующие:
- `HOME` - полное имя домашнего каталога текущего пользователя;
- `PATH` - список имен каталогов, в которых производится поиск команды, при ее указании коротким именем;
- `PS1` - основное приглашение `shell` ко вводу команды;

и т.д.

- Вызов любой команды можно окружить одиночными кавычками `(`), и тогда в соответствующую строку будет подставлен результат стандартного вывода этой команды.
- Среди управляющих конструкций языка содержатся следующие: `for` и `while` для организации циклов, `if` для организации ветвлений и `case` для организации переключателей (естественно, все это специфически приспособлено для работы с текстовыми значениями).

C-shell

Командный язык C-shell главным образом отличается от Bourne-shell тем, что его синтаксис приближен к синтаксису языка Си (это, конечно, не означает действительной близости языков). В основном, C-shell включает в себя функциональные возможности Bourne-shell. Если не вдаваться в детали, то реальными отличиями C-shell от Bourne-shell является поддержка протокола (файла истории) и псевдонимов.

В протоколе сохраняются введенные в данном сеансе работы с интерпретатором командные строки. Размер протокола определяется установкой предопределенной переменной `history`, но последняя введенная командная строка сохраняется всегда. В любом месте текущей командной строки в нее может быть подставлена командная строка (или ее часть) из протокола.

Механизм псевдонимов (alias) позволяет связать с именем полностью (или частично) определенную командную строку и в дальнейшем пользоваться этим именем.

Кроме того, в C-shell по сравнению с Bourne-shell существенно расширен набор предопределенных переменных, а также введены более развитые возможности вычислений (по-прежнему, все значения представляются в текстовой форме).

Korn-shell

Если C-shell является синтаксической вариацией командного языка семейства shell по направлению к языку программирования Си, то Korn-shell - это непосредственный последователь Bourne-shell.

Если не углубляться в синтаксические различия, то Korn-shell обеспечивает те же возможности, что и C-shell, включая использование протокола и псевдонимов.

Реально, если не стремиться использовать командный язык как язык программирования (это возможно, но по мнению автора, неоправданно), то можно пользоваться любым вариантом командного языка, не ощущая их различий.

Команды и утилиты

Что действительно существенно при интерактивной работе в среде ОС UNIX, это знание и умение пользоваться разнообразными утилитами или внешними командами языка shell. Многие из этих утилит являются не менее сложными программами, чем сам командный интерпретатор (и между прочим, командный интерпретатор языка shell сам является одной из утилит, которую можно вызвать из командной строки). В этом разделе мы коротко обсудим, как устроены внешние команды shell и что нужно сделать, чтобы создать новую внешнюю команду.

Организация команды в ОС UNIX

Вообще-то, для создания новой команды не нужно делать почти ничего специального, нужно просто следовать правилам программирования на языке Си. Как известно, каждая правильно оформленная Си-программа начинает свое выполнение с функции main. Эта "полусистемная" функция обладает стандартным интерфейсом, являющимся основой организации команд, которые можно вызывать в среде shell. Внешние команды выполняются интерпретатором shell с помощью связки системных вызовов fork и одного из вариантов exec (см. п. 2.1.6). В число параметров системного вызова exec входит набор текстовых строк. Этот набор текстовых строк передается на вход функции main запускаемой программы.

Более точно, функция main получает два параметра - argc (число передаваемых текстовых строк) и argv (указатель на массив указателей на текстовые строки). Программа, претендующая на ее использование в качестве команды shell, должна обладать точно определенным внешним интерфейсом (параметры обычно вводятся с терминала) и должна контролировать и правильно разбирать входные параметры.

Кроме того, чтобы соответствовать стилю shell, такая программа не должна сама переопределять файлы, соответствующие стандартному вводу, стандартному выводу и

стандартному выводу ошибок. Тогда команде может обычным образом перенаправляться ввод/вывод, и она может включаться в конвейеры.

Перенаправление ввода/вывода и организация конвейера

Как видно из последнего предложения предыдущего пункта, для обеспечения возможностей перенаправления ввода/вывода и организации конвейера при программировании команд не требуется делать ничего специального. Достаточно просто не трогать три начальные дескриптора файлов и правильно работать с этими файлами, а именно, производить вывод в файл с дескриптором `stdout`, вводить данные из файла `stdin` и выводить сообщения об ошибках в файл `stderror`.

Встроенные, библиотечные и пользовательские команды

Встроенные команды представляют собой часть программного кода командного интерпретатора. Они выполняются как подпрограммы интерпретатора, и их невозможно заменить или переопределить. Синтаксис и семантика встроенных команд определены в соответствующем командном языке.

Библиотечные команды составляют часть системного программного обеспечения. Это набор выполняемых программ (утилит), поставляемых вместе с операционной системой. Большинство этих программ (таких как `vi`, `emacs`, `grep`, `find`, `make` и т.д.) исключительно полезно на практике, но их рассмотрение находится за пределами этого курса (по поводу редакторов `vi` и `emacs` и утилиты поддержки целостности программных файлов `make` существуют отдельные толстые книги).

Пользовательской командой является любая выполняемая программа, организованная в соответствии с требованиями, изложенными в п. 5.2.2. Таким образом, любой пользователь ОС UNIX может неограниченно расширять репертуар внешних команд своего командного языка (например, можно написать собственный командный интерпретатор).

Программирование на командном языке

Об этом мы уже говорили. Любой из упоминавшихся вариантов языка `shell` в принципе можно использовать как язык программирования. Среди пользователей ОС UNIX существует много людей, которые пишут на `shell` вполне серьезные программы. Однако по нашему мнению правильнее использовать `shell` для того, для чего он изначально предназначен - для непосредственного интерактивного взаимодействия с системой и для создания не очень сложных командных файлов. Для программирования лучше использовать языки программирования (Си, Си++, Паскаль и т.д.), а не командные языки. Хотя оговоримся, что это личное мнение автора, а выбор способа и стиля программирования является сугубо частным делом каждого программиста.

Средства графического интерфейса пользователей

Хотя многие профессиональные программисты, работающие в среде ОС UNIX, и сегодня предпочитают пользоваться традиционными строчными средствами взаимодействия с системой, широкое распространение относительно недорогих цветных графических терминалов с высоким качеством разрешения привело к тому, что во всех современных вариантах ОС UNIX поддерживаются графические интерфейсы пользователя с системой,

а пользователям предоставляются инструментальные средства для разработки графических интерфейсов с разрабатываемыми ими программами. С точки зрения конечного пользователя средства графического интерфейса, поддерживаемого в разных вариантах ОС UNIX, да и в других системах (например, MS Windows или Windows NT), примерно одинаковы по своему стилю.

Во-первых, во всех случаях поддерживается многооконный режим работы с экраном терминала. В любой момент времени пользователь может образовать новое окно и связать его с нужной программой, которая работает с этим окном как с отдельным терминалом. Окна можно перемещать, изменять их размер, временно закрывать и т.д.

Во-вторых, во всех современных разновидностях графического интерфейса поддерживается управление мышью. В случае ОС UNIX часто оказывается, что обычной клавиатурой терминала пользуются только при переходе к традиционному строчному интерфейсу (хотя в большинстве случаев по крайней мере в одном окне терминала работает один из командных интерпретаторов семейства shell).

В-третьих, такое распространение "мышиного" стиля работы оказывается возможным за счет использования интерфейсных средств, основанных на пиктограммах (icons) и меню. В большинстве случаев, программа, работающая в некотором окне, предлагает пользователю выбрать какую-либо выполняемую ей функцию либо путем отображения в окне набора символических образов возможных функций (пиктограмм), либо посредством предложения многоуровневого меню. В любом случае для дальнейшего выбора оказывается достаточным управления курсором соответствующего окна с помощью мыши.

Наконец, современные графические интерфейсы обладают "дружественностью по отношению к пользователю", обеспечивая возможность немедленного получения интерактивной подсказки по любому поводу. (Наверное, правильнее было бы сказать, что хорошим стилем программирования с использованием графических интерфейсов является стиль, при котором такие подсказки реально обеспечиваются.)

После перечисления всех этих общих свойств современных средств графического интерфейса может возникнуть естественный вопрос: Если в области графических интерфейсов существует такое единообразие, что особенное может быть сказано по поводу графических интерфейсов в среде ОС UNIX? Ответ достаточно прост. Да, конечный пользователь действительно в любой сегодняшней системе имеет дело примерно с одним и тем же набором интерфейсных возможностей, но в разных системах эти возможности достигаются по-разному. Как обычно, преимуществом ОС UNIX является наличие стандартизованных технологий, позволяющих создавать мобильные приложения с графическими интерфейсами.

В этой части курса мы рассмотрим базовый механизм ОС UNIX для организации оконных графических интерфейсов на основе использования разнообразных графических терминалов (оконную систему X), а также два распространенных инструментальных пакета, предназначенных для облегчения разработки графического интерфейса с прикладной программой (индустриальный, используемый сегодня практически во всех вариантах ОС UNIX пакет Motif и свободно распространяемый пакет Tcl/Tk). Каждая из упомянутых тем настолько обширна, что полная информация занимает несколько толстых книг; по поводу каждой из тем можно было бы провести отдельный курс лекций. Поэтому здесь мы приводим только краткий обзор, целью которого является создание общего

представления о методах и средствах организации графического интерфейса в среде ОС UNIX.

Оконная система X как базовое средство графических интерфейсов в среде ОС UNIX

Понятно, что для нормальной организации работы пользовательских программ с графическими терминалами (если учитывать отмеченные выше стандартные требования к графическому интерфейсу) требуется наличие некоторого базового слоя программного обеспечения, скрывающего аппаратные особенности терминала; обеспечивающего создание окон на экране терминала, управление этими окнами и работу с ними со стороны пользовательской программы; дающего возможность пользовательской программе реагировать на события, происходящие в соответствующем окне (ввод с клавиатуры, движение курсора, нажатие клавиш мыши и т.д.). Такой базовый слой графического программного обеспечения принято называть оконной системой.

В мире ОС UNIX предпринималось несколько попыток создания оконных систем, и большинство из них успешно использовалось практически (упомянем, например, оконную систему NeWS компании Sun Microsystems, интерфейс которой основывался на использовании языка Postscript). Однако ни одна из этих систем не выходила за пределы ведомственного использования, что, естественно, резко ограничивало мобильность программ, обладающих графическим интерфейсом. Успеха удалось добиться группе молодых исследователей и программистов из Масачусетского технологического института, которые создали оконную систему под кратким и предельно скромным названием X (кстати, именно так правильно называть систему; по-английски ее грамотно называют не X-Window, а X window system, т.е. "оконная система X"). В настоящее время оконная система X является фактическим стандартом опорных средств графического интерфейса. Система X, дополнительные библиотеки, а также ряд готовых интерфейсных средств распространяются MIT бесплатно (относясь к категории public domain). В то же время сегодня именно оконная система X является базовым механизмом организации графических интерфейсов пользователя в большинстве UNIX-систем.

Общая организация X-Window

Как кажется, оконная система X победила потому, что организация системы очень точно соответствует общей идеологии ОС UNIX. UNIX - это традиционно сетевая операционная система. Девиз Билла Джоя и всей компании Sun Microsystems "The Network is the Computer - Сеть - это компьютер" - в полной мере относится к направлению ОС UNIX в целом. Популярная ныне архитектура организации программно-аппаратных средств "клиент-сервер" всегда была совершенно естественной в мире UNIX. Специализация и разделение функций в сети - это и значит, что для пользователя компьютер и сеть неразличимы.

На этих идеях построена и оконная система X. Поскольку ОС UNIX является интерактивной операционной системой, то каждый работающий в системе пользователь взаимодействует с системой через предоставленный ему терминал (скорее всего, рабочую станцию) и, вообще говоря, вызывает программы, которые будут выполняться на других компьютерах локальной или территориально распределенной сети. Именно эти программы обеспечивают интерфейс с данным пользователем, т.е. они должны иметь возможность работать с терминалом пользователя вне зависимости от того, где они выполняются. Более того, чтобы возможности графического интерфейса этих программ удовлетворяли общим требованиям, нужно, чтобы программы могли не заботиться о

таких деталях, как многооконная организация экрана, текущее расположение окон, управление мышью и т.д.

Оконная система X предоставляет в точности требуемые возможности. На стороне пользовательского терминала находится сервер системы X, обеспечивающий единообразное управление графическим терминалом вне зависимости от его специфических аппаратных характеристик. В других компьютерах сети (которые, фактически, являются серверами с точки зрения организации вычислительного процесса) установлены клиентские части системы X, создающие впечатление у выполняемой программы, что она взаимодействует с локальным терминалом, а на самом деле поддерживающие точно специфицированный протокол взаимодействий с сервером системы X.

Клиентская и серверная части

Как видно из материала предыдущего пункта, клиентская и серверная части оконной системы X, хотя в целом соответствуют идеологии архитектуры "клиент-сервер", обладают тем своеобразием, что серверная часть системы находится вблизи пользователя (т.е. основного клиента вычислительной сети), а клиентская часть системы базируется на мощных серверах сети. Конечно, система X обладает достаточной гибкостью, чтобы допустить расположение серверной и клиентской частей системы в одном компьютере, в разных компьютерах одной локальной сети и удаленных компьютерах, входящих в состав территориально распределенной сети. В зависимости от конфигурации системы X-сервер может обслуживать один или несколько графических экранов, клавиатуру и мышь, реально представляя собой процесс, группу процессов или выделенное компьютерное устройство (X-терминал).

Для обеспечения требуемой гибкости, взаимодействия клиентской и пользовательской частей системы X по мере возможностей не должны были зависеть от используемых сетевой среды передачи данных и сетевых протоколов. Конечно, полная независимость - это теоретически недостижимая цель (всегда и везде), но что касается системы X, то она действительно умеет работать в большинстве распространенных сетевых сред (в том числе, естественно, в стандартных для ОС UNIX сетях, основанных на семействе протоколов TCP/IP).

Основой взаимодействия между клиентом и сервером оконной системы X является так называемый X-протокол, представляющий собой точную спецификацию допустимых запросов от клиента к серверу и допустимых ответов сервера к клиенту. Как указывается в документации оконной системы X, X-протокол обладает следующими особо привлекательными качествами:

- При использовании этого протокола обработка взаимодействий клиента и сервера ведется единообразно, независимо от того, основана она на внутренних механизмах IPC или на реальных сетевых обменах; это позволяет добиться прозрачности сетевой среды как с точки зрения конечного пользователя, так и с точки зрения разработчика прикладных программ.
- За счет наличия строгой и не зависящей от окружения спецификации X-протокол может быть реализован на различных языках в различных операционных средах.
- X-протокол может быть реализован на основе любого надежно поддерживаемого потока байтов (обеспечиваемого внутренними механизмами IPC или внешними сетевыми механизмами); многие из пригодных механизмов являются стандартными и реализованы в большинстве архитектур.

- Для большинства (хотя и не для всех) приложений X-протокол не порождает существенных задержек при работе с графическими терминалами. Обычно задержки вызываются скорее временными потребностями самих терминалов, а не расходами на протокольные взаимодействия клиента и сервера.

Одним из клиентов оконной системы обычно является так называемый "оконный менеджер" (window manager). Это специально выделенный клиент оконной системы, обладающий полномочиями на управление расположением окон на экране терминала. Некоторые из возможностей X-протокола (связанные, например, с перемещением окон) доступны только клиентам с полномочиями оконного менеджера. Во всем остальном оконный менеджер является обычным клиентом.

Базовые библиотеки

Понятно, что теоретически любая прикладная программа, которой требуется взаимодействовать с X-сервером, могла бы работать с ним, обмениваясь сообщениями в соответствии с X-протоколом. Однако, конечно же, это неудобно. Для выполнения любого, самого простого действия с терминалом клиенту требуется обменяться несколькими сообщениями с X-сервером, причем для наиболее распространенных действий последовательность таких сообщений предопределена.

Библиотека Си-функций, которая поставляется вместе с оконной системой X и облегчает взаимодействие Си-программы с X-сервером в соответствии с X-протоколом, называется XLib. Сам X-протокол достаточно компактен, поскольку в нем специфицированы мелкие сообщения, которые, как правило, можно реально использовать только в некоторых комбинациях. XLib - это уже довольно большая библиотека (в документации системы X ее описание занимает отдельный солидный том). Это потому, что каждая функция библиотеки XLib основана на использовании нескольких протокольных сообщений (а после этого общее количество функций XLib определяется законами комбинаторики). Вместе с тем, XLib - это всего-навсего интерфейсная библиотека над X-протоколом. Если не подниматься над уровнем XLib, то для создания любого графического объекта или сценария графического протокола в каждой прикладной программе придется повторять примерно одни и те же последовательности вызовов функций XLib.

Уровнем, который позволяет использовать ранее созданные графические образы и/или заготовки интерфейсов, является библиотека Xt (X Toolkit) Intrinsics. Эта библиотека служит для создания и использования уже существующих элементов пользовательского интерфейса, называемых виджетами (widgets). Виджет - это параметризуемая заготовка части пользовательского интерфейса (кнопка, часть меню, пиктограмма и т.д.), привязываемая к окну экрана терминала. Библиотека Xt Intrinsics выполнена в объектно-ориентированном стиле, так что каждый виджет представляет собой класс, который может использоваться для порождения новых классов, представляющих собой комбинированные виджеты и т.д.

По своим идеям Xt Intrinsics является мощным средством разработки пользовательских графических интерфейсов. Однако эта библиотека разрабатывалась в MIT и в основном являлась исследовательским прототипом. Хотя несколько компаний представили в public domain собственные наборы виджетов, их общее количество оказывается недостаточным для быстрого и качественного производства графических пользовательских интерфейсов. Это позволило занять лидирующее положение на коммерческом рынке инструментальному пакету консорциума Open Software Foundation (OSF) Motif.

Средства разработки графических интерфейсов

Основное назначение тех средств разработки пользовательских графических интерфейсов, которые разрабатываются и поставляются отдельно от оконной системы, является облегчение создания нового графического интерфейса за счет использования существующих параметризованных заготовок. Как видно, в принципе это те же самые идеи, на которых основана объектно-ориентированная библиотека оконной системы X Xt Intrinsics.

И действительно, наиболее распространенный пакет, предназначенный для быстрой и качественной разработки графических пользовательских интерфейсов, Motif, который был спроектирован и разработан в северо-американском консорциуме OSF, в основном является развитием идей Xt Intrinsics. Motif является сугубо коммерческим продуктом. Дело дошло до того, что компания OSF запатентовала внешний интерфейс продуктов, входящих в состав Motif, чтобы не дать кому-нибудь возможность воспроизвести этот интерфейс.

Это привело к настоящему скандалу в сообществе американских программистов, потому что создало опасный прецедент патентования интерфейсов. Если можно закрыть своим авторским правом возможность повторения графического интерфейса, то почему нельзя запатентовать синтаксис языка программирования, интерфейс операционной системы и т.д.? Однако строгость американских законов не оправдывается необязательностью их исполнения, и поэтому недовольные свободолюбивые американские программисты ропчат, но терпят, а тем временем пытаются сагитировать восточно-европейских программистов (не так сильно зависящих от американских законов) на нелегальную свободно доступную реализацию интерфейсов Motif.

С другой стороны, сравнительно недавно (4-5 лет тому назад) в Калифорнийском университете г. Беркли был создан альтернативный механизм под названием Tcl/Tk. Этот механизм основан на наличии специализированного командного языка, предназначенного для описания графических пользовательских интерфейсов, соответствующего интерпретатора и библиотеки ранее разработанных заготовок интерфейсов. Пакет Tcl/Tk распространяется (вместе с полной документацией) свободно, и многие профессиональные программисты находят его более удобным, чем Motif.

Пакет Motif

Motif (официальное название этого продукта - OSF/Motif) представляет собой программный пакет, включающий оконный менеджер, набор вспомогательных утилит, а также библиотеку классов, построенных на основе Xt Intrinsics. Для конечных пользователей оконных систем, опирающихся на Motif, основной интерес представляет менеджер окон, хотя, скорее всего, вы не сможете определить, применяется ли оконный менеджер Motif в используемой вами установке.

Для разработчиков же графических интерфейсов важны все три компонента Motif. Новый интерфейс разрабатывается в графическом же режиме с использованием оконного менеджера. При этом полезно использование утилит Motif и необходимо использование библиотеки классов Motif.

Библиотека классов Motif является расширением библиотеки Xt Intrinsics с целью предания этой библиотеке практического смысла (по-другому можно сказать, что Motif - это то, чем должен был бы быть Xt, если бы при его создании ставились коммерческие

цели). Все графические объекты (правильнее сказать, классы) Xt Intrinsics включаются в библиотеку классов Motif, хотя в ней используются другие имена.

Но Motif существенно расширяет возможности Xt Intrinsics. В его библиотеке поддерживается большое число классов, позволяющих создавать меню, "нажимаемые" кнопки и т.д. Основное назначение этих классов - определение новых виджетов, связанных с окнами.

Однако в Motif поддерживается и новый вид графических объектов (их классов) - так называемые гаджеты (gadgets). Гаджет отличается от виджета тем, что соответствующий класс также может использоваться для создания элементов интерфейса, но графический объект не привязывается к определенному окну. При отображении на экран гаджета используется окно объекта, относящегося к суперклассу класса гаджета.

Понятно, что здесь мы не можем привести подробное описание Motif (еще раз повторим, что соответствующий материал содержится в нескольких солидных книгах). Однако основная идея должна быть понятна: развитая библиотека классов языка Си++, возможности применения этих классов при использовании обычного стиля программирования и поддержка визуального программирования с немедленным отображением получающихся графических объектов.

Язык и интерпретатор Tcl/Tk

Продукт Tcl/Tk в действительности представляет собой два связанных программных пакета, которые совместно обеспечивают возможность разработки и использования приложений с развитым графическим пользовательским интерфейсом. Название Tcl относится к "командному языку инструментальных средств - tool command language", и, как не странно, его рекомендуется произносить "тикл". Это простой командный язык для управления приложениями и расширения их возможностей. Язык Tcl является "встраиваемым": его интерпретатор реализован в виде библиотеки функций языка Си, так что интерпретатор может быть легко пристыкован к любой прикладной программе, написанной на языке Си.

Tk (рекомендуемое произношение - "ти-кей") является библиотекой Си-функций, ориентированной на облегчение создания пользовательских графических интерфейсов в среде оконной системы X (т.е., по сути дела, некоторый аналог Xt Intrinsics). С другой стороны, аналогично тому, как это делается в командных языках семейства shell, функции библиотеки Tk являются командами языка Tcl, так что любой программист может расширить командный репертуар языка Tcl путем написания новой функции на языке Си.

Совместно, Tcl и Tk обеспечивают четыре преимущества для разработчиков приложений и пользователей (мы используем здесь авторские тексты разработчиков). Во-первых, наличие командного языка Tcl дает возможность в каждом приложении использовать мощный командный язык. Все, что требуется от разработчика приложения, чтобы удовлетворить его/ее специфические потребности, - это создать несколько новых команд Tcl, требующихся приложению (и, возможно, другим приложениям - явно традиционный стиль командного программирования в ОС UNIX). После этого нужно связать прикладную программу с интерпретатором Tcl и пользоваться полными возможностями командного языка.

Вторым преимуществом использования Tcl/Tk является возможность быстрой разработки графических интерфейсов. Многие интересные оконные приложения могут быть

написаны в виде скриптов языка Tcl без привлечения языков Си или Си++ (а Tcl позволяет скрыть многие несущественные детали). Как утверждают разработчики Tcl/Tk, пользователи оказываются способными к созданию новых графических интерфейсов уже после нескольких часов знакомства с продуктом. Другой особенностью языка Tcl, способствующей быстрой разработке оконных приложений, является то, что язык является интерпретируемым. Можно опробовать новую идею интерфейса, выражающуюся в сотнях или тысячах строк кода на языке Tcl, без потребности вызова новых программных средств, путем простого нажатия на клавишу мыши (не наблюдая существенных задержек при использовании современных рабочих станций).

Третьим преимуществом языка Tcl является то, что его можно применять в качестве языка "склейки" приложений. Например, любое основанное на Tcl и использующее Tk оконное приложение может направить свой скрипт любому другому аналогично ориентированному приложению. С использованием Tcl/Tk можно создавать приложения, работающие в стиле мультимедиа, и опять же они смогут обмениваться скриптами, поскольку пользуются общим интерпретатором командного языка Tcl и общей внешней библиотекой Tk.

Наконец, четвертым удобством интегрированного пакета Tcl/Tk является удобство пользователей. Для написания нового приложения в среде Tcl/Tk достаточно выучить несколько совершенно необходимых команд, и этого окажется достаточно. Другими словами оказывается возможным инкрементальный (пошаговый) стиль погружения в предмет. Такая возможность всегда радует сердце и греет душу.

Впрочем, заметим, что далеко не все программисты разделяют выраженное выше глубоко радостное отношение разработчиков Tcl/Tk к своему продукту.

Современное состояние ОС UNIX

Операционная система UNIX, являющаяся первой в истории мобильной ОС, обеспечивающей надежную среду разработки и использования мобильных прикладных систем, одновременно представляет собой практическую основу для построения открытых программно-аппаратных систем и комплексов. Именно широкое внедрение в практику ОС UNIX позволило перейти от лозунга Открытых Систем к практической разработке этой концепции. Большой вклад в развитие направления Открытых Систем внесла деятельность по стандартизации интерфейсов ОС UNIX.

Тем не менее, до сих пор можно выделить несколько ветвей ОС UNIX, различающихся не только реализацией, но временами интерфейсами и семантикой (хотя, по мере развития процесса стандартизации, эти различия становятся все менее значительными). В приводимом ниже кратком обзоре мы затрагиваем только некоторые варианты ОС UNIX, которые, по нашему мнению, наиболее существенны в настоящее время.

UNIX System V Release 4 и UnixWare

Канонические исходные тексты ОС UNIX, как известно, были написаны сотрудниками телефонной компании AT&T, и долгое время авторские права, равно как и права на продажу лицензий на использование исходных текстов принадлежали этой компании. В дальнейшем, по причине технических сложностей с разработкой и сопровождением сложного программного продукта и некоторых юридических затруднений компания AT&T образовала дочернюю компанию USL (UNIX System Laboratories) с основной задачей развития и сопровождения исходных текстов ОС UNIX.

Именно USL выпустила вариант ОС UNIX System V 4.0, который стал фактическим стандартом операционной системы UNIX и явился основой многочисленных версий ОС UNIX, производимых поставщиками рабочих станций и серверов. Последним успехом USL как дочерней компании AT&T явился выпуск SVR 4.2. Помимо прочего, в этой ОС был впервые в истории UNIX реализован механизм легковесных процессов (threads), работающих на общей виртуальной памяти и позволяющих использовать аппаратные возможности так называемых "симметричных мультипроцессорных архитектур", в которых несколько процессоров имеют равноправный доступ к общей оперативной памяти.

В 1993 году компания USL была поглощена компанией Novell, и в настоящее время фактически является подразделением этой компании. При этом владение торговой маркой UNIX было передано консорциуму X/Open. В 1994 году USL в составе Novell была почти незаметна; видимо, сказывались необходимые структурные, организационные и маркетинговые преобразования. Однако в начале 1995 года компания Novell объявила о выпуске нового варианта своей ОС UnixWare 2.0, основанного на System V 4.2, что свидетельствует о завершении процесса внедрения USL в Novell.

Компания Novell приобрела широкую известность и заработала основной капитал на рынке локальных сетей персональных ЭВМ. Распространенная "сетевая" ОС NetWare на самом деле всего лишь обеспечивает сетевой доступ персональных компьютеров, работающих под управлением MS-DOS, к ресурсам серверов (главным образом, файловых). Возрастающие возможности компьютеров, основанных на процессорах компании Intel, их фактический переход из класса персональных компьютеров в класс развитых рабочих станций, недостаточные возможности ОС типа MS-DOS для эффективного использования этих компьютеров заставили компанию Novell обратить внимание на операционную систему UNIX.

Первая версия системы под названием UnixWare целиком основывалась на SVR 4.0, но включала ряд расширений, специфичных для Novell. Следует отметить, что многие пользователи этой системы были не слишком ей довольны: она была не очень надежна и сложно администрировалась. В начале 1995 года появился релиз UnixWare 2.0, основанный на SVR 4.2. По отзывам пользователей эта система гораздо более продвинута. В частности, обеспечивается полный графический интерфейс администратора, файловая система очень надежна, допускается доступ к файлам, хранимым на серверах NetWare и т.д. В конце 1995 года компания Novell обещает выпустить новый продукт, который будет основываться на UNIX, но при этом будет поддерживать все функции NetWare.

Системы, основанные на System V Release 4

На базе System V возникло много коммерческих компаний. Их продукты мы кратко рассмотрим в этом разделе.

Solaris компании Sun Microsystems

Известно, что в течении многих лет основой операционных систем (SunOS) компании Sun являлся UNIX BSD. Однако, начиная с SunOS 4.0, произошел полный переход на System V 4.0. Это связано, прежде всего, с тем, что SVR 4.0 включает функциональные возможности UNIX линии BSD.

Sun Microsystems внесла ряд существенных расширений в SVR 4.0. Прежде всего это касается обеспечения распараллеливания программ при использовании симметричных

мультипроцессорных компьютеров (механизм потоков управления - threads). В SVR 4.0 этот механизм отсутствовал (он появился только в SVR 4.2), а компания Sun уже активно выпускала мультипроцессорные компьютеры. Поэтому в SunOS был реализован собственный механизм threads, что потребовало многочисленных переделок в ядре системы.

Solaris является внешней оболочкой SunOS и дополнительно включает средства графического пользовательского интерфейса и высокоуровневые средства сетевого взаимодействия (в частности, средства вызова удаленных процедур - RPC). Заметим, что хотя самая первая реализация механизма RPC принадлежит компании Xerox, именно реализация Sun стала фактическим стандартом и лицензирована многими компаниями.

HP/UX компании Hewlett-Packard, DG/UX компании Data General, AIX компании IBM

HP/UX, DG/UX и AIX обладают многими отличиями. В частности, в этих версиях ОС UNIX поддерживаются разные средства генерации графических пользовательских интерфейсов (хотя все они основаны на использовании оконной системы X), по-разному реализованы threads и т.д.

Однако все эти системы объединяет тот факт, что в основе каждой из них находится SVR 4.x. Поэтому основной набор системных и библиотечных вызовов в этих реализациях совпадает.

Заметим, что в компании IBM существовал план разработки полностью самостоятельной реализации AIX на основе микроядра. Однако в последнее время IBM отказалась от этой идеи, хотя собственное микроядро (новая реализация микроядра Mach) уже было создано.

Santa Cruz Operation и SCO UNIX

Варианты ОС UNIX, производимые компанией SCO и предназначенные исключительно для использования на Intel-платформах, до сих пор базируются на лицензированных исходных текстах System V 3.2. Однако SCO довела свои продукты до уровня полной совместимости со всеми основными стандартами (в тех позициях, для которых существуют стандарты).

Консерватизм компании объясняется прежде всего тем, что ее реализация ОС UNIX включает наибольшее количество драйверов внешних устройств и поэтому может быть установлена практически на любой Intel-платформе. Естественно, при переходе на другой вариант опорных исходных текстов ядра системы могла бы потребоваться массовая переделка драйверов.

Тем не менее, SCO имеет соглашение с французской компанией Chorus Systems о разработке новой версии SCO UNIX, базирующейся на микроядре Chorus и предназначенной для использования в системах реального времени.

Open Software Foundation и OSF-1

OSF была первой коммерческой компанией, решившейся на полную реализацию ОС UNIX на базе микроядра Mach. Результатом этой работы явилось создание ОС OSF-1. Как

утверждают, OSF-1 на самом деле не является полностью лицензионно чистой системой: в ней используется часть исходных текстов SVR 4.0.

На сегодняшний день наиболее серьезным потребителем OSF-1 является компания Digital Equipment на своих платформах, основанных на микропроцессорах Alpha. В OSF-1 поддерживаются все основные стандарты ОС UNIX, хотя многие утверждают, что пока система работает не очень устойчиво.

Свободно распространяемые и коммерческие варианты ОС UNIX семейства BSD

Многие годы варианты ОС UNIX, разработанные в Калифорнийском университете г. Беркли, являлись реальной альтернативой AT&T UNIX. Например, ОС UNIX BSD 4.2 была бесплатно доступна в исходных текстах и достаточно широко использовалась даже в нашей стране на оригинальных и воспроизведенных машинах линии DEC. BSD 4.3 являлась основой популярной ОС Ultrix компании DEC. UNIX BSD использовался в SunOS, и т.д.

Группа BSD оказала огромное влияние на общее развитие ОС UNIX. До появления SVR 4.0 проблемой для пользователей являлась несовместимость наборов системных вызовов BSD и System V. Как мы отмечали выше, в SVR 4.0 был реализован общий набор системных вызовов.

Около 5 лет назад в Беркли была начата работа над микроядерной реализацией BSD 4.4. Работа была уже близка к завершению, когда компания USL, являвшаяся в то время владельцем исходных текстов System V, подала в суд на университет Беркли, мотивируя это тем, что в BSD 4.4 нелегально используются части исходных текстов SVR 4.0. Процесс продолжался около двух лет и закончился победой Беркли, хотя в то же время было выставлено условие произвести полную очистку текстов BSD от следов System V. Все это, естественно, затормозило выпуск BSD 4.4, полный вариант которой до сих пор недоступен.

Несколько лет назад группа BSD разделилась на коммерческую и некоммерческую части. Новая коммерческая компания получила название BSDI. Обе подгруппы выпустили варианты ОС UNIX для Intel-платформ под названиями 386BSD и BSD386, причем коммерческий вариант был гораздо более полным.

Сегодня популярен новый свободно распространяемый вариант ОС UNIX, называемый FreeBSD. Ведутся работы над более развитыми версиями BSDNet.

Другие свободно распространяемые варианты ОС UNIX

Системы семейства BSD на сегодняшний день не являются единственными свободно доступными вариантами ОС UNIX. Ниже мы коротко опишем два других варианта.

Linux университета Хельсинки

LINUX - это оригинальная реализация ОС UNIX для Intel-платформ, выполненная молодым сотрудником университета Хельсинки Линусом Торвальдом.

Ядро системы написано в традиционной технологии (т.е. без использования микроядра). Однако по отзывам любителей ядро LINUX отличается высоким качеством кода и

хорошей модульностью. Кроме того, утверждается, что при аккуратном программировании прикладные программы, созданные в среде LINUX, без особых проблем переносятся в среду коммерческих систем, базирующихся на System V.

LINUX распространяется свободно, является очень экономичной ОС (т.е. не требует слишком много ресурсов компьютера) и весьма популярен среди молодежи. Практически каждую неделю появляется новый драйвер, работающий в LINUX. Этой ОС посвящена одна из самых активных телеконференций в Internet. Уже издается несколько регулярных журналов, связанных исключительно с тематикой LINUX.

Hurd Free Software Foundation

Проект системы Hurd явился попыткой довести до логического завершения знаменитый проект GNU Ричарда Столлмана, основателя и президента Фонда свободного программного обеспечения (Free Software Foundation - FSF). Общей целью проекта GNU является создание полномасштабной свободно распространяемой мобильной программной среды, совместимой с соответствующими коммерческими продуктами, но существенно превосходящей их по своим возможностям. FSF уже в течение многих лет распространяет высококачественные программные средства: компиляторы, отладчики, редакторы и т.д. Однако собственного ядра операционной системы у разработчиков проекта GNU не было.

Основной идеей проекта Hurd было использование в качестве основы системы готового варианта микроядра Mach, бесплатно распространяемого университетом Карнеги-Меллон. Более подробно технологию Hurd мы рассмотрим в п. 8.4.3, а пока заметим, что уже год назад система была близка к уровню бета-тестирования, однако до сих пор ее выпуск не объявлен. Сам Ричард Столлман рекомендует пока использовать LINUX совместно с продуктами линии GNU.

Стандарты ОС UNIX

До тех пор, пока господствовала узкая трактовка ОС UNIX (т.е. пока ОС UNIX не была коммерческим продуктом), не было потребности в стандартизации средств этой операционной системы. Немногочисленные высококвалифицированные пользователи ОС UNIX сами могли разобраться в особенностях и отличиях используемой версии системы и выбрать то подмножество ее средств, которое обеспечивало переносимость разрабатываемого приложения.

Однако, с выходом ОС UNIX на коммерческий рынок, переходом к широкой трактовке системы и существенным увеличением числа пользователей различных ее вариантов, стало необходимым ввести хотя бы возможность производства основанных на ОС UNIX операционных систем, которые были бы действительно совместимы. Для этого необходима стандартизация (интерфейсов) средств операционной системы на разных уровнях. Такая работа ведется уже около 10 лет, еще не завершена и вряд ли когда-либо будет завершена в виде окончательного набора стандартов де-юре. Тем не менее, даже полученные результаты позволяют производителям обеспечить пользователей разных аппаратных платформ операционными системами, достаточно удобными для использования и позволяющими разрабатывать мобильные прикладные системы, которые могут выполняться на компьютерах, оснащенных операционными системами с аналогичными свойствами.

Прежде чем перечислить наиболее важные официальные и фактические стандарты, принимаемые во внимание производителями систем, основанных на ОС UNIX, сформулируем, что же понимается под стандартом интерфейсов ОС. Стандарт интерфейсов ОС - это обычно сводка более или менее формальных синтаксических (интерфейсных) и семантических (поведенческих) свойств специфицируемых средств операционной системы.

System V Interface Definition (SVID)

Одним из наиболее ранних стандартов де-факто ОС UNIX явился изданный UNIX System Laboratories (USL) одновременно с выпуском версии ОС UNIX System V Release 4 документ System V Interface Definition (SVID). Если кратко напомнить историю, то владельцем оригинальных исходных текстов ОС UNIX являлась компания AT&T Bell Laboratories (именно работники этой компании Ритчи, Томпсон и Керниган разработали в начале 1970-х самый первый мобильный вариант ОС UNIX). В 1980-е годы компания AT&T основала дочернюю компанию USL, которой были переданы права на исходные тексты и торговую марку ОС UNIX. USL выпустила системы с System V R.4.0 до System V R.4.2, после чего в конце 1993 г. была поглощена компанией Novell, ставшей владельцем исходных текстов ОС UNIX (под давлением общественности торговая марка "UNIX" была передана компании X/Open). (По поводу самых последних изменений см. раздел 7.1.)

Несмотря на все эти пертурбации SVID продолжает существовать и пользоваться авторитетом у производителей. Как кажется, главным объяснением этому является тот факт, что сегодня большинство коммерческих вариантов ОС UNIX основаны на лицензированных у AT&T-USL-Novell исходных текстах UNIX. Поэтому не очень сложно полностью удовлетворять этому фактическому стандарту. Естественно, SVID как документ, изданный одной компанией без его предварительного общественного обсуждения, никогда не будет принят в качестве официального стандарта.

Деятельность комитетов POSIX

Следует вспомнить, что наряду с версиями ОС UNIX, развивавшимися в компании AT&T (затем в USL, затем в Novell, затем...), исторически существовало еще направление BSD (Berkeley Standard Distribution), успешно поддерживаемое небольшой, но всемирно известной группой из университета г. Беркли (шт. Калифорния). В свое время (в конце 1970-х) университет получил из AT&T исходные тексты 16-разрядной ОС UNIX, на основе которых была произведена 32-разрядная система, которая сначала использовалась на компьютерах семейства VAX, а затем была перенесена на многие другие аппаратные платформы. В результате наборы системных вызовов UNIX AT&T и BSD стали значительно различаться.

Хотя большинство коммерческих реализаций UNIX основывалось на System V, UNIX BSD всегда был популярен в университетах, и общественность потребовала определения некоторого интерфейса, который являлся бы по сути объединением средств AT&T и BSD. Эта работа была начата Ассоциаций профессиональных программистов Открытых Систем UniForum, а затем продолжена в специально созданных рабочих группах POSIX (Portable Operating System Interface). В рабочих группах POSIX разрабатываются многие стандарты открытых систем, но наиболее известным и авторитетным является принятый ISO по представлению IEEE стандарт POSIX 1003.1, в котором определены минимальные требуемые средства операционной системы (по сути дела, UNIX).

Деятельность X/Open

Международная организация X/Open, которая выполняет многие работы, связанные с пропагандой и анализом использования открытых систем, кроме того, собирает и систематизирует де-юре и де-факто стандарты, имеющие промышленное значение, в так называемом X/Open Common Application Environment (CAE). Спецификаций интерфейсов средств, входящих в CAE, публикуются в многотомном документе X/Open Portability Guide (XPG).

Стандарт ANSI C

Очень важным в мире UNIX является принятый сначала ANSI, а потом и ISO международный стандарт языка программирования Си. Дело в том, что в этом стандарте специфицирован не только непосредственно язык Си, но и библиотеки, необходимые в каждой стандартной реализации. Поскольку с самого своего появления язык Си и соответствующие системы программирования были неразрывно связаны с ОС UNIX, то состав стандартных библиотек достаточно точно соответствует стандартной среде ОС UNIX.

Другие стандарты

Перечисленные четыре стандарта, только два из которых являются официально принятыми, наиболее авторитетны для производителей операционных систем, претендующих на совместимость с ОС UNIX. Особенностью этих стандартов является их полная машинная независимость.

Имеется другая разновидность стандартов де-факто, распространяемых на некоторый класс аппаратных архитектур. Примером такого стандарта может служить документ, принятый международной организацией SPARC International документ SPARC Complience Definition, содержащий машинно-зависимые уточнения к машинно-независимым спецификациям интерфейсов. Аналогичный документ разрабатывался организацией 88/Open, связанной с RISC-процессорами фирмы Motorola.

Среди других индустриальных де-факто стандартов для современных вариантов ОС UNIX наиболее важны фактический стандарт оконной системы, поддерживаемый X Cosortium, в основе которого находится лаборатория Масачусетского технологического института (MIT), являющаяся разработчиком системы X, а также спецификации интерфейсов инструментального средства разработки графических пользовательских интерфейсов OSF/Motif, разработанные в Open Software Foundation (OSF).

Кроме того, заметим, что в OSF разработан документ OSF Application Environment Specification (AES), содержащий спецификации интерфейсов ОС OSF/1, являющей собственной реализацией OSF ОС UNIX на базе новой микроядерной технологии (правда, до сих пор в этой реализации используются фрагменты исходного текста System V). AES является расширением SVID, POSIX 1003.1 и XPG.

Перспективные ОС, поддерживающие среду ОС UNIX

Микроядро - это минимальная стержневая часть операционной системы, служащая основой модульных и переносимых расширений. По-видимому, большинство операционных систем следующего поколения будут обладать микроядрами. Однако

имеется масса разных мнений по поводу того, как следует организовывать службы операционной системы по отношению к микроядру: как проектировать драйверы устройств, чтобы добиться наибольшей эффективности, но сохранить функции драйверов максимально независимыми от аппаратуры; следует ли выполнять операции, не относящиеся к ядру, в пространстве ядра или в пространстве пользователя; стоит ли сохранять программы имеющихся подсистем (например, UNIX) или лучше отбросить все и начать с нуля.

В широкий обиход понятие микроядра ввела компания Next, в операционной системе которой использовалось микроядро Mach. Небольшое привилегированное ядро этой ОС, вокруг которого располагались подсистемы, выполняемые в режиме пользователя, теоретически должно было обеспечить небывалую гибкость и модульность системы. Но на практике это преимущество было несколько обесценено наличием монолитного сервера, реализующего операционную систему UNIX BSD 4.3, которую компания Next выбрала в качестве оболочки микроядра Mach. Однако опора на Mach дала возможность включить в систему средства передачи сообщений и ряд объектно-ориентированных сервисных функций, на основе которых удалось создать элегантный интерфейс конечного пользователя с графическими средствами конфигурирования сети, системного администрирования и разработки программного обеспечения.

Следующей микроядерной операционной системой была Windows NT компании Microsoft, в которой ключевым преимуществом использования микроядра должна была стать не только модульность, но и переносимость. (Заметим, что отсутствует единодушное мнение по поводу того, следует ли на самом деле относить NT к микроядерным ОС.) ОС NT была построена таким образом, чтобы ее можно было применять в одно- и мультипроцессорных системах, основанных на процессорах Intel, Mips и Alpha (и тех, которые придут вслед за ними). Поскольку в среде NT должны были выполняться программы, написанные для DOS, Windows, OS/2 и систем, совместимых со стандартами Posix, компания Microsoft использовала присущую микроядерному подходу модульность для создания общей структуры NT, не повторяющей ни одну из существующих операционных систем. Каждая операционная система эмулируется в виде отдельного модуля или подсистемы.

Позднее микроядерные архитектуры операционных систем были объявлены компаниями Novell/USL, Open Software Foundation (OSF), IBM, Apple и другими. Одним из основных конкурентов NT в области микроядерных ОС является Mach 3.0, система, созданная в университете Карнеги-Меллон, которую как IBM, так и OSF взялись довести до коммерческого вида. (Компания Next в качестве основы для NextStep пока использует Mach 2.5, но тоже внимательно присматривается к Mach 3.0.) Другим конкурентом является микроядро Chorus 3.0 компании Chorus Systems, выбранное USL в качестве основы новых реализаций ОС UNIX. Некоторое микроядро будет использоваться в SpringOS фирмы Sun, объектно-ориентированном преемнике ОС Solaris (если, конечно, Sun доведет работу над SpringOS до конца). Очевидна тенденция к переходу от монолитных к микроядерным системам (хотя, как мы отмечали в предыдущем разделе, этот процесс не является прямолинейным: компания IBM сделала шаг назад и отказалась от перехода к микроядерной технологии). Кстати, это совсем не новость для компаний QNX Software Systems и Unisys, которые уже в течение нескольких лет выпускают пользующиеся успехом микроядерные операционные системы. ОС QNX пользуется спросом на рынке систем реального времени, а CTOS фирмы Unisys популярна в области банковского дела. В обеих системах успешно использована модульность, присущая микроядерным ОС.

Понятие микроядра

Микроядро реализует базовые функции операционной системы, на которые опираются другие системные службы и приложения. Основной проблемой при конструировании микроядерной ОС является распознавание тех функций системы, которые могут быть вынесены из ядра. Такие важные компоненты ОС как файловые системы, системы управления окнами и службы безопасности становятся периферийными модулями, взаимодействующими с ядром и друг с другом.

Когда-то казалось, что многоуровневая архитектура ядра ОС UNIX является вершиной в области конструирования операционных систем. Основные функциональные компоненты операционной системы - файловая система, взаимодействие процессов (IPC - interprocess communications), ввод-вывод и управление устройствами - были разделены на уровни, каждый из которых мог взаимодействовать только с непосредственно примыкающим к нему уровнем.

Несмотря на неплохие практические результаты такая структура теперь все больше воспринимается монолитной, поскольку вся операционная система связана иерархией уровней. Множественность и нечеткость интерфейсов между уровнями затрудняет модификацию системы; для этого требуется хорошее знание операционной системы, масса времени и элемент везения.

В микроядерных архитектурах вертикальное распределение функций операционной системы заменяется на горизонтальное. Компоненты, лежащие выше микроядра, используют средства микроядра для обмена сообщениями, но взаимодействуют непосредственно. Микроядро лишь проверяет законность сообщений, пересылает их между компонентами и обеспечивает доступ к аппаратуре.

Это свойство микроядерных систем позволяет естественно использовать их в распределенных средах. При получении сообщения микроядро может его обработать или переслать другому процессу. Поскольку для микроядра безразлично, поступило ли сообщение от локального или удаленного процесса, подобная схема передачи сообщений является удобной основой удаленных вызовов процедур (RPC - Remote Procedure Calls). Однако пересылка сообщений производится медленнее обычных вызовов функций; оптимизация пересылки сообщений является критическим фактором успеха микроядерной операционной системы. Например, в ОС Windows NT в некоторых случаях для оптимизации используется разделяемая память. Расходы на дополнительную фиксированную память микроядра оправдываются повышением эффективности передачи сообщений.

Поскольку вся машинно-зависимая часть ОС изолирована в микроядре, для переноса системы на новый процессор требуется меньше изменений и эти изменения логически сгруппированы. При имеющемся разнообразии на рынке процессоров способность операционной системы работать на разных процессорах является единственной возможностью убедить пользователей покупать новые машины.

Расширяемость также является необходимым свойством современных операционных систем. В отличие от аппаратных средств, которые устаревают за несколько лет, операционные системы могут с пользой эксплуатироваться в течение десятилетий. В жизни каждой операционной системы настает момент, когда в нее требуется внести функции, не заложенные в исходную конструкцию. Микроядерная организация операционных систем позволяет добиться возможности производства управляемых и

надежно работающих расширений на основе ограниченного набора четко определенных интерфейсов микроядра.

В действительности, правильнее говорить не только о расширяемости, но и о масштабируемости микроядерных ОС с возможностью получения варианта операционной системы, в наилучшей степени соответствующей особенностям аппаратной платформы и прикладной области. Микроядерная организация ОС позволяет легко добиться и этого качества.

Одной из проблем традиционно организованных операционных систем является наличие множества интерфейсов прикладного программирования (API - Application Programming Interface), не все из которых хорошо документированы. В результате невозможно гарантировать правильность программ, использующих несколько API, и даже правильность работы самой операционной системы.

Микроядро, обладающее небольшим набором API (микроядро OSF обеспечивает около 200 системных вызовов, а крохотное микроядро QNS - всего лишь 14), увеличивает шансы получения качественных программ. Конечно, этот компактный интерфейс облегчает жизнь только системных программистов; прикладной программист по прежнему должен бороться с сотнями вызовов.

Основным принципом организации микроядерных ОС является включение в состав микроядра только тех функций, которые абсолютно необходимо выполнять в режиме супервизора и в защищенной памяти. Обычно в микроядро включаются машинно-зависимые программы (включая поддержку мультипроцессорной работы), некоторые функции управления процессами, обработка прерываний и поддержка пересылки сообщений.

Во многих случаях в микроядро включается функция планирования процессов, но в реализации Mach компании IBM планировщик процессов размещен вне микроядра, а микроядро используется только для непосредственного управления процессами. Конечно, при этом требуется тесное взаимодействие внешнего планировщика и входящего в состав ядра диспетчера.

В некоторых реализациях (например, в реализации OSF) в микроядро помещаются драйверы устройств. В реализациях IBM и Chorus драйверы размещаются вне микроядра, но для регулирования режимов разрешения и запрещения прерываний часть программы драйвера выполняется в пространстве ядра. В NT драйверы устройств и другие функции ввода-вывода выполняются в пространстве ядра, но реально используют ядро только для перехвата и передачи прерываний. Следует заметить, что оба подхода допускают динамическое подключение драйверов к системе и их отключение.

Однако имеются другие доводы в пользу выделения драйверов из состава микроядра. Например, поскольку во многих случаях драйверы могут не зависеть от особенностей аппаратуры, такой подход облегчает переносимость системы.

Микроядро Mach университета Карнеги-Меллон

В разрабатывавшейся компанией IBM ОС Workplace (теперь она отказалась от завершения этой ОС) использовалось микроядро Mach 3.0, расширенное в кооперации с OSF средствами поддержки параллельной обработки и реального времени. Микроядро заведовало функциями взаимодействия процессов, управления виртуальной памятью,

управления процессами и нитями (threads), управления процессорами (включая мультипроцессорные системы), а также управления вводом-выводом и обработки прерываний. Файловая система, планировщик процессов, сетевые службы и система безопасности вынесены из микроядра. В IBM эти компоненты называют PNS (Personally Neutral Services), поскольку они используются во всех эмуляторах операционных систем.

Управление процессами и нитями в Workplace являлись функцией ядра. Но на самом деле в ядре был расположен только диспетчер процессов. Планировщик, ведающий приоритетами, определяющий порядок выполнения и заказывающий диспетчеризацию процессов и нитей, функционировал вне ядра.

Управление памятью также распределялось между микроядром и PNS. Ядро управляло аппаратурой страничной памяти. Подсистема управления страничной памятью, работающая вне ядра, определяла стратегию замещения страниц. Подобно планировщику эта подсистема являлась заменяемым компонентом.

На уровне PNS могут располагаться не только такие внутренние подсистемы, как файловая система и драйверы устройств, но и сетевые службы и даже системы управления базами данных. По мнению IBM, размещение подобных служб в непосредственной близости от микроядра позволит повысить их эффективность за счет сокращения числа вызовов функций и возможности использовать собственные драйверы устройств.

OSF ориентируется на массивно параллельные аппаратные системы. Активно изучаются вопросы изменения поведения операционной системы при возрастании числа процессоров. В такой системе микроядро Mach будет работать на всех процессорах, а сервер OSF/1 потребуется только на некоторых из них.

Как планирует OSF, в будущих версиях OSF/1 на основе Mach будет поддерживаться возможность размещения сервера OSF/1 в пространстве ядра или в пользовательском пространстве в соответствии с выбором системного администратора при конфигурировании системы. Выполнение сервера OSF/1 в пространстве ядра позволит повысить производительность, так как вместо передачи сообщений будут использоваться вызовы процедур, и сервер всегда будет целиком располагаться в памяти. При выполнении сервера в пользовательском пространстве будет возможен его свопинг, что потенциально увеличит память, доступную для программ пользователя. Заметим, что примерно такой же подход используется USL в версиях UNIX, основанных на микроядре Chorus. Системные функции будут разработаны и отлажены в пользовательском пространстве, а потом можно будет перенести в пространство ядра для достижения наилучшей производительности.

Микроядро Chorus компании Chorus Systems

Микроядро Chorus во многих отношениях походит на реализации Mach, выполненные IBM и OSF. Chorus включает поддержку распределенных процессоров, нескольких распределенных серверов операционной системы (во многом похожую на комбинацию Mach-OSF/1), управления памятью и обработки прерываний. Поддерживается также прозрачное взаимодействие с другими экземплярами микроядра Chorus, что делает Chorus хорошей основой для сильно распределенных систем.

Драйверы устройств не включаются в ядро. Аналогично подходу IBM, драйверы получают доступ к аппаратуре через ядро. Это дает возможность компоненту более

высокого уровня - менеджеру устройств, отслеживать работу драйверов, функционирующих в разных узлах распределенной системы.

Примеры микроядерных реализаций ОС UNIX

Коротко охарактеризуем некоторые варианты ОС UNIX, построенные на основе технологии микроядра.

OSF-1 компании Open Software Foundation

ОС OSF/1 1.3 основана на микроядре Mach. IBM является членом OSF, и эти компании обменивались технологиями организации микроядра. Однако по некоторым важным направлениям подходы IBM и OSF различаются. В версии 1.3 весь сервер OSF/1 работает в пользовательском пространстве и использует функции Mach.

Почему же OSF решилась на микроядерную реализацию монолитного сервера UNIX? Как говорят специалисты, OSF, OSF/1 является слишком хорошей и надежной системой, чтобы можно было ее бросить и начать все сначала. В OSF/1 1.3 используется более 90% кода предыдущих версий OSF/1. С другой стороны, чтобы улучшить возможности управления объектами, часть ядра Mach была переписана на Си++.

В результате OSF/1 1.3 получилась не такой модульной, как ОС Workplace. Но использовав значительную часть OSF/1, компания OSF смогла раньше IBM получить более или менее полную микроядерную реализацию системы.

MiX компании Chorus Systems

Существует несколько реализаций микроядра Chorus. Chorus/MiX, версия компании Chorus операционной системы с интерфейсами UNIX, включает отдельные версии, совместимые с SVR3.2 и SVR4. USL собирается объявить Chorus/MiX V.4 микроядерной реализацией SVR4. USL и Chorus Systems планируют совместную работу по разработке Chorus/MiX V.4 в качестве будущего направления UNIX. Специально для использования на персональных компьютерах компания Chorus поддерживает реализацию Chorus/MiX, совместимую с SCO.

Hurd Free Software Foundation

Операционная система Hurd на протяжении последних нескольких лет разрабатывается в Фонде свободного программного обеспечения (Free Software Foundation). По своему замыслу ОС Hurd должна была явиться последней точкой в реализации проекта GNU - проекта полной свободно распространяемой совместимой с ОС UNIX среды.

В числе основных разработчиков FSF исторически не было специалистов по внутренней организации операционных систем. В частности, поэтому при реализации Hurd был выбран подход, основанный на предоставленной университетом Карнеги-Меллон версии микроядра Mach, а также использовании готовой файловой системы из Висконсинского университета. Над микроядром в пользовательском режиме дописан набор серверов, которые, однако, в отличие от OSF1 и MiX, не реализуют напрямую возможностей системных вызовов UNIX. Реализация аналога системных вызовов выполнена в виде набора библиотечных подпрограмм, выполняемых в адресных пространствах пользовательских процессов.

ОС Hurd еще не выпущена в свет, хотя уже более года назад в ее среде работал shell, emacs, GCC и другие компоненты программного обеспечения GNU. Кроме того, пока Hurd будет доступен только на платформах Intel.