

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ УКРАИНЫ

Национальный технический университет
“Харьковский политехнический институт”

А.С.Дервянко, М.Н.Солощук

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Часть II

Обзор операционных систем

Учебное пособие

Харьков 2002

ББК 32 - 973.018

УДК 681.3.06

Рецензенти: **Г.І.Загарій**, д-р техн. наук, проф., завідувач кафедри “Автоматика і комп’ютерні системи управління” Української державної академії залізничного транспорту; **З.В.Дудар**, канд. техн. наук, проф., завідувач кафедри “Програмне забезпечення ЕОМ” Харківського національного університету радіоелектроніки.

Интеллектуальная собственность НТУ “ХПИ”. Все права защищены.

Операционные системы: Деревянко А.С., Солощук М.Н. Учебное пособие. - Харьков: НТУ “ХПИ”, 2002. - ...с.

Во второй части учебного пособия рассматривается применения общих принципов управления ресурсами в конкретных операционных системах. Приведено описание большого количества современных операционных систем и их анализ с точки зрения концепций управления ресурсами. Предназначено для студентов и специалистов направлений “Компьютерные науки” и “Компьютерная инженерия”

ISBN

У другій частині навчального посібника розглядається застосування загальних принципів керування ресурсами у конкретних операційних системах. Наведено опис великої кількості сучасних операційних систем і їх аналіз з точки зору концепцій керування ресурсами. Призначається для студентів та спеціалістів напрямків “Комп’ютерні науки” та “Комп’ютерна інженерія”

Илл. - 54, библиогр. - 42 назв.

© А.С. Деревянко, М.Н. Солощук, 2002

© Национальный технический университет “ХПИ”, 2002

Содержание

Введение

Глава 1. Операционная система MS DOS

- 1.1. История и архитектура
- 1.2. Управление программами
- 1.3. Управление памятью
- 1.4. Ввод-вывод и файловая система
- 1.5. Другие свойства MS DOS

Контрольные вопросы

Глава 2. Операционная система Windows 3.x

- 2.1. История и архитектура
- 2.2. Управление процессами
- 2.3. Управление памятью
- 2.4. Другие свойства Windows 3.x

Контрольные вопросы

Глава 3. Операционные системы Windows 9x/ME

- 3.1. История и архитектура
- 3.2. Управление процессами
- 3.3. Управление памятью
- 3.4. Файловая система
- 3.5. Другие свойства Windows 9x

Контрольные вопросы

Глава 4. Операционная система OS/2

- 4.1. История и архитектура
- 4.2. Многозадачность
- 4.3. Управление памятью
- 4.4. Устройства и файловая система
- 4.5. Средства взаимодействия
- 4.6. Другие свойства OS/2

Контрольные вопросы

Глава 5. Операционные системы Windows NT/2000

- 5.1. История и архитектура
- 5.2. Ядро и планирование процессов
- 5.3. Адресные пространства
- 5.4. Ввод-вывод
- 5.5. Процессы-серверы
- 5.6. Система безопасности
- 5.7. Файловая система NTFS

Контрольные вопросы

Глава 6. Семейство операционных систем Unix

- 6.1. История и современное состояние
- 6.2. Архитектура Unix
- 6.3. Процессы
- 6.4. Нити
- 6.5. Планирование процессов
- 6.6. Управление памятью
- 6.7. Средства взаимодействия процессов
- 6.8. Файловые системы
- 6.9. Интерфейсы Unix
- 6.10. Unix-системы фирмы Caldera

Контрольные вопросы

Глава 7. Операционные системы "тонких" клиентов

- 7.1. Карманные персональные компьютеры
- 7.2. Операционная система PalmOS
- 7.3. Операционная система Windows CE
- 7.3. Новые тенденции встроенных ОС

Контрольные вопросы

Глава 8. Операционные системы MacOS и MacOS X

- 8.1. Компьютеры Apple
- 8.2. Операционная система Mac OS
- 8.3. Операционная система Mac OS X

Контрольные вопросы

Глава 9. Операционная система BeOS

- 9.1. Короткая история и позиционирование системы
- 9.2. Потоки и команды
- 9.3. Средства взаимодействия
- 9.4. Управление памятью
- 9.5. Образы
- 9.6. Устройства и файловые системы

Контрольные вопросы

Глава 10. Операционная система QNX

- 10.1. Архитектура
- 10.2. Управление процессами
- 10.3. Средства взаимодействия
- 10.4. Файловая система
- 10.5. Управление устройствами

- 10.6. Сетевые взаимодействия
- 10.7. Графическая система Photon
- Контрольные вопросы

Глава 11. Вычислительная система AS/400

- 11.1. Архитектура
- 11.2. Объекты
- 11.3. Управление памятью
- 11.4. Программы и процессы
- 11.5. Постоянные объекты и ввод-вывод
- 11.6. Система безопасности
- 11.7. Новые свойства и перспективы
- Контрольные вопросы

Глава 12. Операционные системы мейнфреймов

- 12.1. История и архитектура мейнфреймов
- 12.2. Операционная система VSE/ESA
- 12.3. Операционная система z/OS
- 12.4. Операционная система z/VM
- Контрольные вопросы

Глава 13. Платформа Java как операционная среда

- 13.1. Основные свойства платформы Java
- 13.2. Виртуальная машина Java
- 13.3. Многопоточность и синхронизация
- 13.4. Управление памятью в куче
- 13.5. Защита ресурсов
- 13.6. JavaOS и Java для тонких клиентов
- 13.7. Перспективы технологий Java
- Контрольные вопросы

Заключение

Литература

Введение

В первой части учебного пособия были рассмотрены общие концепции управления вычислительными ресурсами. Во второй части учебного пособия мы рассматриваем реализации этих концепций в конкретных современных ОС.

Материалы этой части компоновались по следующему принципу.

В главах 1 – 6 представлены ОС, которые должны быть более или менее знакомы нашему читателю. Эти ОС имеют или имели достаточно широкое распространение в нашей стране. ОС в пределах этих глав упорядочены "от простого к сложному": от меньших объемов ресурсов к большим.

В главах 7 – 12 мы рассматриваем (также от простого к сложному) ОС, которые являются несколько экзотическими для отечественного читателя. Однако динамика развития большинства из этих ОС позволяет предположить, что в скором времени они получат должное представительство и у нас.

Наконец, глава 13 имеет к ОС лишь косвенное отношение, но показывает применение тех же принципов в программном обеспечении, лежащем на более высоком уровне, чем ОС.

Хотя мы и старались отразить в нашем обзоре последние версии рассматриваемых ОС, это не было нашей основной целью. Более важным нам представлялось изложение основных тех архитектурных принципов построения той или иной ОС, которые почти не меняются от версии к версии.

По той же причине мы включили в обзор несколько ОС, которые на сегодняшний день могут считаться устаревшими. Относительно таких ОС следует иметь в виду, что, во-первых, устаревшая ОС, даже не

поддерживаемая более производителем, может оставаться в эксплуатации еще довольно долго, во-вторых, концепции или даже части программного кода устаревшей ОС могут использоваться ее наследниками. Так например, пока эта книга готовилась к печати, прекратила существование компания Be, но следует ожидать, что решения, появившиеся в BeOS будут использованы в следующих версиях PalmOS.

Глава 1. Операционная система MS DOS

1.1. История и архитектура

ОС MS DOS была разработана фирмой Microsoft по заказу IBM для недавно появившихся персональных компьютеров IBM PC. Первая версия ОС появилась в 1980 г. IBM PC и его программное обеспечение не рассматривались тогда фирмой IBM как возможное стратегическое направление, отсюда и MS DOS представляет собой ОС с минимальными возможностями. MS DOS прошла долгий путь развития, но это развитие заключалось прежде всего в приспособливании ОС к опережающему росту возможностей аппаратуры и, в меньшей степени – в совершенствовании структуры самой ОС и развитии ее принципиальных возможностей.

MS DOS в основе своей была и остается однозадачной, однопользовательской системой [3]. Ядро системы разработано для 16-разрядного процессора Intel 8086, следовательно, не использует защищенный режим и объем памяти свыше 1 Мбайт, ставшие доступными в следующих моделях.

Архитектура MS DOS показана на рисунке 1.1. Можно говорить о том, что системное программное обеспечение ПЭВМ состоит из двух уровней. Нижний уровень составляет Базовая Система Ввода-Вывода (BIOS), хранимая в ПЗУ. Второй уровень составляет собственно MS DOS. (Можно сказать, что BIOS является компонентом аппаратной части ПЭВМ, но последующие ОС на платформе Intel-Pentium почти не используют функции BIOS). Системные вызовы реализованы в программных прерываниях. Всего возможно 256 типов (кодов) прерываний. Из них прерывания с 16-ричными кодами от 0 до F зарезервированы за аппаратурой, прерывания с кодами от 10 до 1F –

обращения к BIOS, прерывания с кодами от 20 до 3F – обращения к MS DOS. По-видимому, изначально предполагалось, что непосредственно работать с аппаратурой будет только BIOS, MS DOS будет обращаться к BIOS для выполнения операций на аппаратуре, а приложения – только к MS DOS. Однако в последующих версиях MS DOS перехватывает все больше функций BIOS. Приложениям доступны не только любые обращения к MS DOS и к BIOS, но и такие команды, которые в других системах являются привилегированными, например, команды ввода-вывода, следовательно, приложения имеют доступ к аппаратуре в обход ОС и BIOS.

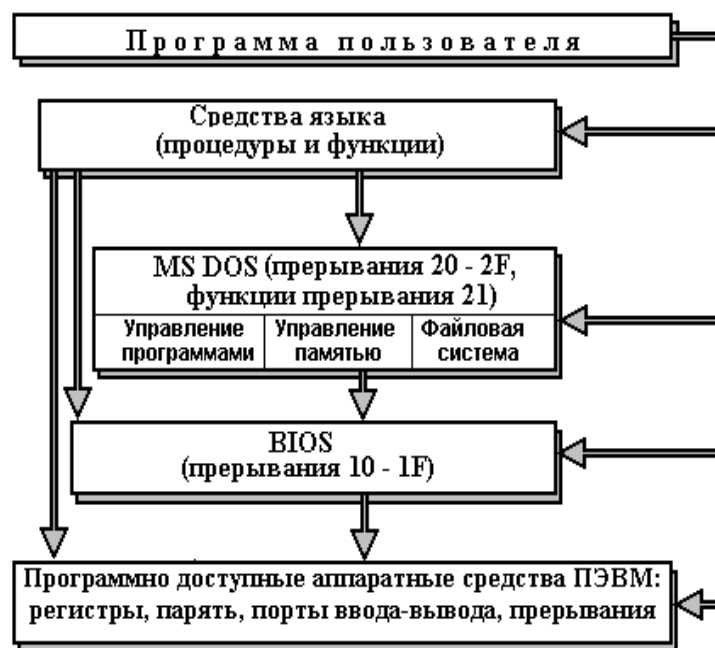


Рисунок 1.1 Архитектура MS DOS

1.2. Управление программами

Как было сказано, MS DOS является однозадачной ОС. Роль блока контекста выполняемой программы играет структура данных,

предваряющая программный сегмент и именуемая Префиксом Программного Сегмента (PSP). PSP формируется системой при загрузке программы, ОС сохраняет среди своих данных адрес PSP выполняемой программы, который играет роль системного идентификатора активного процесса.

Программы в MS DOS бывают двух видов: COM- и EXE-программы. COM-программа не может иметь размер более 64 Кбайт. Во всех программах адресация ведется относительно содержимого базовых регистров. В COM-программе содержимое всех базовых регистров одинаково и формируется Загрузчиком ОС. В EXE-программе содержимое базовых регистров может меняться, поэтому при загрузке такой программы происходит настройка адресов – модификация адресных полей тех команд программы, которые выполняют загрузку базовых регистров.

Поскольку программы пользователя имеют возможность перехватывать любые – программные и аппаратные – прерывания, пользователь имеет возможность создавать резидентные программы – программы, которые после завершения остаются в памяти. Резидентная программа обычно содержит в себе обработчик перехваченного прерывания (например, от таймера, от клавиатуры), который выполняет активизацию резидентной программы по этому прерыванию. И создание резидентных программ, и перехват прерываний поддерживаются системными вызовами MS DOS. После выполнения резидентной программой своих действий возобновляется выполнение прерванной программы. Таким образом, резидентные программы в MS DOS обеспечивают некоторое подобие многозадачности. Сохранение/восстановление контекста (регистров) прерванной программы отчасти выполняется механизмом команд INT (программное прерывание) и RET (возврат из прерывания), отчасти возлагается на резидентную программу. Для полного переключения контекста резидентная программа

должна найти в системной области память адрес PSP прерванной программы и заменить его на адрес своего PSP, но многие резидентные программы этого не делают и выполняются в контексте прерванной программы. Отметим также, что все системные вызовы MS DOS совместно используют только два стека и, таким образом, являются нереентерабельными. Поэтому на применение системных вызовов в резидентных программах накладываются значительные ограничения.

1.3. Управление памятью

Управление памятью оперирует блоками переменной длины в реальной памяти. Виртуальная адресация в программе – относительно содержимого одного или нескольких сегментных регистров. Если вся программа (код, стек, данные) помещается в пределах одного 64-разрядного сегмента, то занесение реального адреса начала сегмента в сегментные регистры производится загрузчиком MS DOS. Если же программа многосегментная, она содержит команды загрузки сегментных регистров, загрузчик модифицирует эти команды реальными адресами сегментов.

Структура адресного пространства MS DOS показана на рисунке 1.2. Программы и пользовательские данные размещаются в области, обозначенной как "распределяемая память". При загрузке программы ей выделяются два блока памяти, называемые сегментом окружения и программным сегментом. Программа в ходе выполнения может запрашивать/освобождать любое количество дополнительных блоков памяти. Единицей распределения памяти является параграф (16 байт). Выделяемый блок памяти всегда состоит из целого числа параграфов. Первый параграф каждого блока содержит Блок Управления Памятью (MSCB), в котором среди прочего содержится идентификатор программы –

владельца блока или признак свободного блока. Поле размера является завуалированным указателем на следующий блок: адрес следующего блока можно определить, зная адрес текущего и его размер. В системе не предусмотрены никакие средства борьбы с фрагментацией памяти, так как в однозадачной ОС интенсивность запросов на выделение/освобождение памяти не может быть слишком большой.



Рисунок 1.2 Распределение памяти в MS DOS

1.4. Ввод-вывод и файловая система

Управление вводом-выводом осуществляется драйверами устройств. Структура драйвера обладает некоторой избыточностью, так как формировалась, по-видимому, в расчете на многозадачность в будущем, которая так и не состоялась. Драйверы загружаются и инициализируются при загрузке системы. Обращения к драйверам унифицированы, и каждый драйвер "умеет" выполнять лишь подмножество операций из единого для всех драйверов набора возможных операций. Различаются драйверы блочных (дисковых) и символьных устройств.

Логическая файловая система MS DOS обеспечивает иерархическую структуру хранения данных в виде "леса" – отдельного дерева каталогов на

каждом логическом диске. Имя файла состоит из собственно имени (8 символов) и расширения (3 символа). Физическая структура хранения данных на жестком диске показана на рисунке 1.3. Структура информации на гибком диске соответствует структуре одного логического раздела.

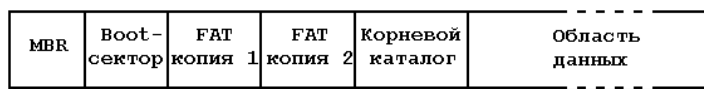


Рисунок 1.3 Структура информации на диске файловой системы FAT

Первый сектор диска занимает Главная Загрузочная Запись (MBR – Master Boot Record), которая содержит программу начальной загрузки и таблицу разделов – информацию о разбиении физического диска на логические. В таблице разделов предусмотрены позиции для 4 разделов, но каждый раздел может содержать свое расширение MBR, то есть может быть разбит еще на 4 раздела и т.д. MBR не является структурой, относящейся к MS DOS, она загружается программой начальной загрузки BIOS и применяется со всеми ОС, работающими на платформе Intel-Pentium. Более того, разные логические диски могут содержать разные ОС. Программа начальной загрузки в MBR определяет, с какого логического диска должна загружаться ОС и считывает загрузочный сектор этого диска.

Загрузочный сектор содержит программу начальной загрузки ОС (если диск является загрузочным) и информацию о параметрах логического диска).

Центральной структурой файловой системы MS DOS является Таблица Размещения Файлов (FAT – File Allocation Table). FAT представляет собой "карту" дискового пространства области данных. Область данных условно разбивается на кластеры – участки из целого числа смежных секторов. Размер кластера фиксирован для данного

логического диска и кластер является единицей распределения дисковой памяти. Каждому кластеру соответствует элемент FAT. Размер элемента – 12 или 16 бит, отсюда файловую систему часто называют FAT12 или FAT16. В каждом элементе каталога, описывающем файл или подкаталог, содержится номер первого кластера, выделенного файлу. В элементе FAT, соответствующем этому кластеру находится номер следующего кластера и т.д. Специальный код в элементе FAT индицирует последний кластер файла. Специальные коды элементов зарезервированы для описания свободных и сбойных кластеров. Поскольку FAT является ключевой структурой файловой системы, на диске для надежности хранятся две ее копии.

Следующая структура – корневой каталог – содержит описание файлов и подкаталогов в корневом каталоге. Корневой каталог (а также и подкаталоги) состоит из 32-байтных элементов, в каждом из которых содержится имя и расширение, атрибуты, размер, номер начального кластера файла или подкаталога. Все структуры, показанные на рисунке 1.3, создаются при форматировании диска, следовательно, размер корневого каталога ограничен. Подкаталоги же создаются в области данных по мере необходимости, их размер практически неограничен.

1.5. Другие свойства MS DOS

Поскольку MS DOS спроектирована как однозадачная система, средства взаимодействия процессов в ней почти не предусматриваются. Для взаимодействия резидентных программ применяются прерывания: резидентная программа перехватывает обработку прерывания с определенным номером. В спецификациях MS DOS предусмотрены два прерывания, которые можно использовать для этой цели (2D, 2F), однако,

многие приложения используют для этого "незанятые" прерывания с номерами, большими 3F.

Командный процессор MS DOS – программа COMMAND.COM – обеспечивает выполнение команд и утилит, связанных с управлением файловой системой, выполнением программ, управлением параметрами окружения. Обеспечивается также перенаправление стандартного ввода-вывода и конвейерное выполнение команд. COMMAND.COM – процессор командной строки, повсеместно MS DOS используется с полноэкранными оболочками, наиболее популярная из которых – Norton Commander.

MS DOS послужила основой для целого ряда совместимых с ней ОС, наиболее развитая из которых – PC DOS фирмы IBM. Развитие самой MS DOS закончилось на версии 6, версия 7 существует только как встроенная в Windows 9x. PC DOS продолжает развиваться и сейчас существует в версии 8, в которую опционно включены, например, невытесняющая многозадачность и графический интерфейс. Несмотря на явную бесперспективность развития MS DOS, ее применение будет продолжаться, прежде всего, в качестве ОС клиентского рабочего места с минимальными вычислительными ресурсами. Огромное число приложений, существующих для MS DOS, диктует для других ОС необходимость обеспечивать ту или иную эмуляцию среды MS DOS для выполнения этих приложений.

Контрольные вопросы

1. Обладает ли MS DOS какими-либо преимуществами перед персональными ОС следующего поколения?

2. В чем состоит разница между COM- и EXE-программами?
3. Каким образом в MS DOS может быть обеспечено подобие многозадачности?
4. Какая из описанных в Части I моделей памяти применяется в MS DOS?
5. Из каких соображений в MS DOS не включены средства борьбы с фрагментацией оперативной памяти?
6. Почему память программы MS DOS ограничивается 640 Кбайтами?

Глава 2. Операционная система Windows 3.x

2.1. История и архитектура

Операционная система Windows 1.x (1987 г.) разрабатывалась фирмой Microsoft по заказу IBM прежде всего для того, чтобы подготовить пользователей к графическому интерфейсу, который должен был появиться в новой версии совместно разрабатываемой этими двумя фирмами OS/2. Но значительный успех на рынке, пришедший с версией Windows 3.0, явился одной из причин того, что фирма Microsoft решила разорвать свое сотрудничество с IBM и сосредоточиться на этой линии продуктов.

Строго говоря, Windows 1.x - 3.x представляет собой не самостоятельную ОС, а надстройку над MS DOS [2]. Windows запускается как приложение MS DOS, которое перехватывает у MS DOS управление некоторыми ресурсами — прежде всего памятью, процессами и символьными устройствами. Управление же файловой системой остается за MS DOS и для выполнения соответствующих функций Windows обращается к MS DOS.

Архитектура Windows 3.x показана на рисунке 2.1. Ядро Windows состоит из трех модулей, из которых: Kernel – обеспечивает системные функции, User – объекты интерфейса пользователя, GDI – графические функции.

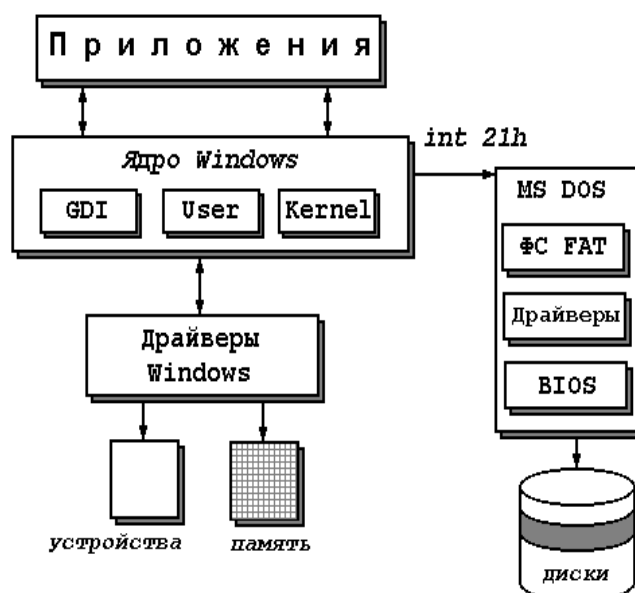


Рисунок 2.1 Архитектура Windows 3.x

Windows первоначально разрабатывалась для процессора Intel 80286, а затем – и для Intel 80386, поэтому ядро и драйверы Windows работают в защищенном режиме. Драйверы устройств Windows уже не используют функции BIOS, а самостоятельно управляют внешними устройствами на физическом уровне. (Это не относится к дисковым драйверам, так как функции файловой системы остаются за MS DOS).

2.2. Управление процессами

Windows 3.x обеспечивает невытесняющую многозадачность. В среде Windows могут выполняться одновременно несколько программ-процессов, но активный процесс не может быть прерван системой, переключение процессов может происходить только в том случае, если

текущий активный процесс переходит в состояние ожидания. Отсюда программирование в Windows является событийно-управляемым. Программа представляет собой цикл, каждая итерация которого начинается с ожидания какого-либо сообщения (каковым может быть, например, нажатие кнопки мыши), затем сообщение обрабатывается, и следует новая итерация цикла, которая начинается с ожидания нового сообщения. Ожидание сообщения в начале каждого цикла и является тем моментом, когда процесс может быть вытеснен. Если же программа находится в бесконечном цикле, в теле которого нет системных вызовов, включающих в себя ожидание, то она не может быть прервана и блокирует работу всей системы.

2.3. Управление памятью

Как было сказано, Windows первоначально разрабатывалась для процессора Intel 80286, в котором механизм динамической трансляции адресов работает в сегментной модели виртуальной памяти и используется 16-разрядное адресное слово. Поэтому адресное пространство процесса состоит из сегментов размером не более 64 Кбайт каждый, хотя общее виртуальное адресное пространство процесса может достигать 4 Гбайт. API управления памятью позволяет выделять/освобождать сегменты, а также управлять перемещаемостью и вытесняемостью сегментов. При адаптации к процессору Intel 80386 (сегментно-страничная модель и 32-разрядное адресное слово) версии Windows для этого процессора стали использовать страничную часть механизма трансляции адресов для создания общего для всех процессов виртуального страничного пространства, размер которого в 4 раза превышает размер реальной памяти, а в этом страничном пространстве память распределяется сегментами. API продолжает оставаться сегментным, а свопинг ведется на

страничном уровне. Имеется две группы вызовов API – для управления локальными и глобальными сегментами.

В Windows используется динамическая компоновка программ во время загрузки. Модули динамической компоновки, в том числе и системные, совместно используются выполняющимися программами.

2.4. Другие свойства Windows 3.x

Хотя приоритет в создании полноэкранного графического интерфейса пользователя принадлежит фирме Apple (она даже пыталась в судебном порядке оспорить правомочность использования этой идеи фирмой Microsoft), именно в Windows с таким интерфейсом познакомилась основная масса пользователей. Каждый процесс в Windows выполняется в собственном окне, имеющем типовые элементы управления. Для представления объектов используется иконика, многие операции над объектами выполняются при помощи мыши. Однако, интерфейс Windows 3.x не является объектно-ориентированным, так как не обеспечивает автоматического связывания с каждым объектом определенного набора свойств и методов. Средства графического интерфейса неотъемлемо встроены в ядро системы.

Средства взаимодействия процессов – общие сегменты памяти и сообщения. Сообщения посылаются от одного процесса другому или порождаются внешними событиями. В системе имеется общая очередь сообщений, обслуживаемая только по дисциплине FIFO, что снижает надежность системы, так как если процесс не выбирает адресованное ему сообщение, оно блокирует всю очередь.

Поскольку Windows 3.x использует защищенный режим процессора, она обеспечивает значительно лучшую защиту ресурсов, чем MS DOS. Вместе с тем, Windows 3.x является однопользовательской системой, т.е.,

не включает в себя средств аутентификации пользователей и, соответственно, авторизации. Последняя версия – Windows for Workgroups 3.11 расширена средствами совместного использования ресурсов локальной сети (файлы, принтеры) группами пользователей.

Windows 3.x, по-видимому, можно считать бесперспективной системой, так как объем парка ПЭВМ той мощности, для которой применение Windows 3.x может быть оптимальным, неуклонно уменьшается. Применяются компьютеры либо с совсем малым объемом ресурсов (MS DOS), либо с большим их объемом (Windows 9x). Для Windows 3.x было создано большое число приложений и поэтому среда Windows 3.x поддерживается в более поздних ОС (Windows 9x/NT/2000, OS/2). Но в настоящее время процесс вытеснения этих приложений новыми версиями, разработанными для современных ОС, уже почти завершен.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Почему Windows 3.x нельзя считать полноценной ОС, а только надстройкой над MS DOS?
2. Как изменялась модель памяти Windows 3.x в ходе ее эволюции?
3. Как в Windows 3.x реализуется программирование, управляемое событиями?
4. Каким образом активный процесс в Windows 3.x может лишиться ресурса центрального процессора?
5. Как Windows 3.x выполняет управление файлами?
6. Почему системные модули Windows 3.x могут быть нереентерабельными?

Глава 3. Операционные системы Windows 9x/ME

3.1. История и архитектура

ОС Windows 95 разработана фирмой Microsoft прежде всего для предотвращения перехода пользователей Windows 3.x в среду OS/2 с увеличением мощности персональных вычислительных средств. Windows 95 справилась с этой задачей, но не за счет своих объективных достоинств, а за счет крупномасштабной рекламной компании, начавшейся задолго до ее появления на рынке.

В отличие от Windows 3.x, Windows 95 является полнофункциональной ОС, осуществляющей управление всеми ресурсами вычислительной системы [5]. Для ее выполнения не требуется MS DOS, код MS DOS v.7 встроен в Windows 95 только для поддержки выполнения приложений MS DOS.

Архитектура Windows 95 показана на рисунке 3.1. ОС Windows 98 [15] (и Windows ME) отличаются от Windows 95 прежде всего дизайном интерфейса и некоторыми системными сервисами, но строятся на принципиально том же ядре.

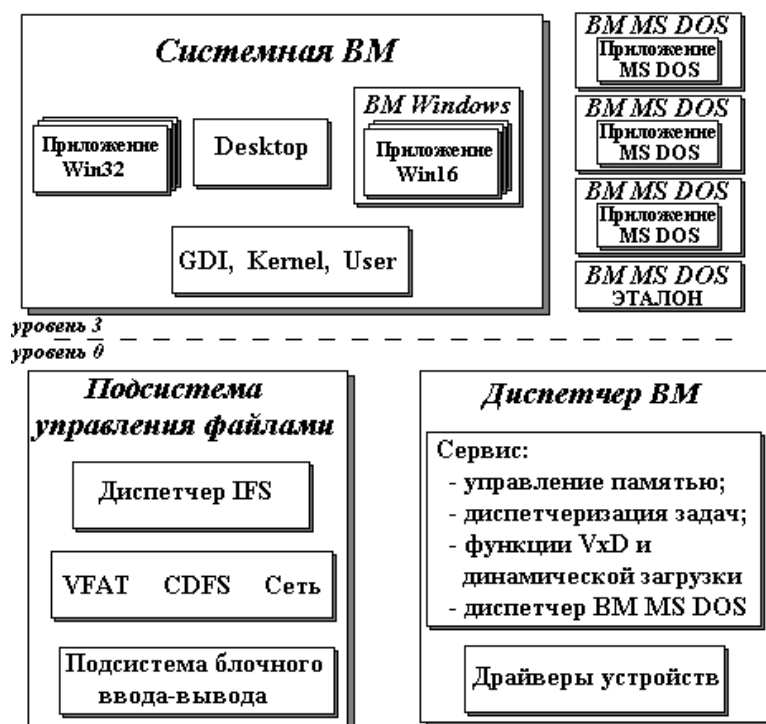


Рисунок 3.1 Архитектура Windows 95

Windows 95 разработана для процессора не ниже Intel 80386, то есть ориентирована на 32-разрядное машинное слово и защищенный режим процессора. Основу той части Windows 95, которая работает в незащищенном режиме, составляет так называемая системная виртуальная машина, которая обеспечивает среду выполнения всех приложений, написанных для Windows 95. Функции модулей GDI, Kernel, User – те же, что и в Windows 3.x, около 50% объема этих модулей просто перенесены из Windows 3.x.

В среде системной виртуальной машины также работает виртуальная машина Windows. Эта виртуальная машина полностью эмулирует Windows 3.x с ее 16-разрядной адресацией, режимом невытесняющей многозадачности, общей очередью и т.д., и приложения Windows 3.x работают в ее среде.

Для выполнения приложений MS DOS создается виртуальная машина MS DOS как копия эталонной машины MS DOS. Каждая такая

виртуальная машина MS DOS обеспечивает адресное пространство для выполнения программ MS DOS и содержит код MS DOS v.7. Выполнение низкоуровневых операций обеспечивается для виртуальной машины MS DOS частью ОС Windows 95, работающей в режиме ядра (уровень 0). Приложения MS DOS независимы друг от друга, но они совместно используют драйверы реального режима и разделяют младшие адреса памяти.

Часть системы, работающая в режиме ядра, обеспечивает собственно управление ресурсами: памятью, процессами и т.д.

3.2. Управление процессами

Windows 95 обеспечивает вытесняющую многозадачность, то есть разделение процессорного времени между двумя и более процессами и нитями. Однако, поскольку часть кода ядра ОС заимствована из Windows 3.x и является нереентерабельной, прерывания активного процесса возможны не в любой момент времени. Для совместимости с заимствованным нереентерабельным кодом в Windows 95 введены специальные модули-переходники, которые, во-первых, преобразуют 32-разрядный API в 16-разрядный, а, во-вторых, блокирует совместное использование нереентерабельной части ядра. Планирование процессов выполняется с динамическим перевычислением приоритетов. Механизм управления процессами обеспечивается двумя модулями:

- основной диспетчер, который перевычисляет приоритеты через каждые 20 мсек (этот интервал называется квантом);
- диспетчер квантования, который распределяет время внутри кванта.

Квант отдается процессу с наивысшим на данный момент приоритетом. Если несколько процессов имеют высший приоритет, то квант делится между ними поровну.

Всего имеется 32 градации приоритетов. Правила изменения приоритетов следующие:

- внешнее событие (нажатие клавиши или кнопки мыши) повышает приоритет;
- повышенный таким образом приоритет постепенно снижается до исходного (если не наступает новое событие);
- приоритет процесса, который захватывает монопольный ресурс (например, дисковый ввод-вывод) повышается, это повышение отменяется, когда ресурс освобождается.

3.3. Управление памятью

Адресное пространство процесса (речь идет о процессах-приложениях, разработанных для Windows 95) структурировано таким образом, как показано на рисунке 3.2.

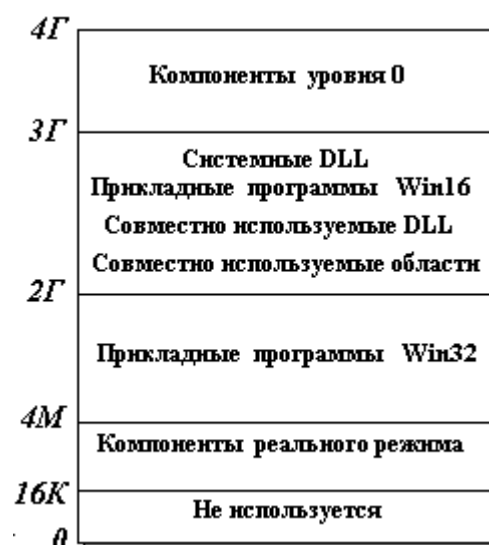


Рисунок 3.2 Структура адресного пространства для приложения Win32

- область адресного пространства 0 - 16 Кбайт не адресуется; эта область используется драйверами реального режима MS DOS;
- область 16 Кбайт - 4 Мбайт не используется процессом (она используется компонентами реального режима);
- область 4 Мбайт - 2 Гбайт составляет частное адресное пространство процесса, в ней размещаются локальные коды и данные процесса;
- область 2 Гбайт - 3 Гбайт – совместно используемые объекты: библиотеки динамической компоновки, в том числе и системные, относящиеся к кольцу 3, такие, как Kernel, GDI и USER; все объекты, совместно используемые разными процессами, причем такой объект имеет один и тот же виртуальный адрес во всех его использующих процессов; в этом адресном пространстве также выполняются приложения Win16;
- в области 3 Гбайт - 4 Гбайт размещаются компоненты системы, работающие в реальном режиме, в том числе VxD, подсистемы управления виртуальными машинами и файлами.

Каждому процессу в Windows 95 выделяется всего один сегмент и виртуальный адрес, таким образом, представляет собой только смещение в сегменте. Для процесса, таким образом, обеспечивается плоская модель памяти. Элементы страничных каталогов, относящиеся к адресному пространству выше 2 Гбайт, для всех процессов указывают на одни и те же таблицы страниц второго уровня. Обращение процесса к системе имеет вид вызова процедуры, находящейся в адресном пространстве процесса (в старших адресах памяти). Совместное использование верхней половины виртуального адресного пространства всеми процессами создает потенциальную (и часто реализующуюся в Windows 9x) возможность нарушений в работе системы: если один процесс "портит" содержимое

верхней части своего адресного пространства, то он тем самым "портит" его и для всех других процессов.

Совместно используемые области памяти, размещаемые в адресном пространстве 2 - 3 Гбайт, называются в Windows 9x "файлами, отображаемыми в память". Один процесс создает такой файл, другой – открывает его. Системные вызовы создания/открытия "файла, отображаемого в память" возвращают виртуальный адрес общей области памяти (одинаковый для всех процессов, его использующих).

3.4. Файловая система

В Windows 95 применяется механизм устанавливаемой (загружаемой) файловой системы, который обеспечивает системы работать с томами, содержащими различную организацию данных. Структура устанавливаемой файловой системы показана на рисунке 3.3.

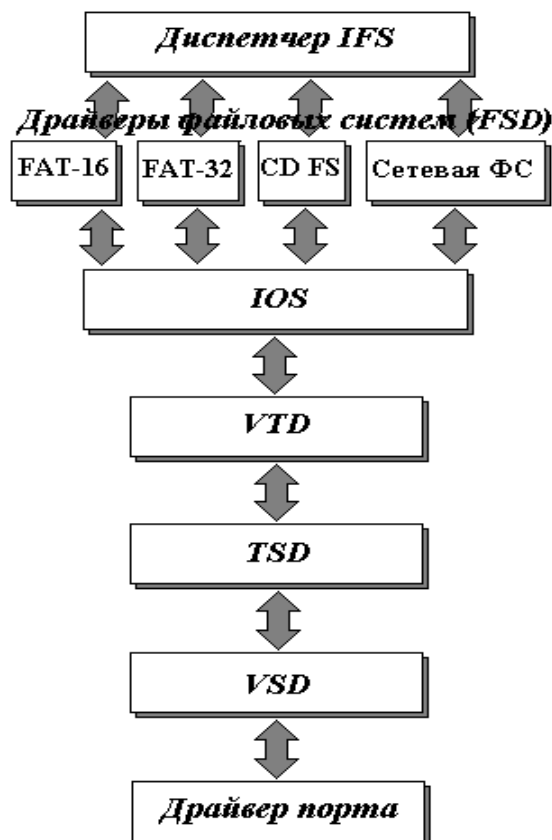


Рисунок 3.3 Инсталлируемая файловая система Windows 9x

Самый верхний уровень файловой системы Диспетчер Инсталлируемой Файловой Системы (ISF – Installable File System), который обеспечивает интерфейс между запросами процессов и конкретной файловой системой. Диспетчер IFS играет роль абстрактной файловой системы. Конкретные файловые системы называются драйверами файловых систем ФС (FSD – File System Driver). Драйверы FSD ответственны за семантику конкретных файловых систем. Каждый такой FSD поддерживает определенную организацию файловых систем и обслуживает запросы, которые передает ему Диспетчер IFS. Диспетчер IFS обеспечивает также для FSD некоторые общие сервисные функции: управление памятью, управление событиями, синтаксический разбор имен файлов и т.п. Каждый FSD при своей инициализации регистрируется у Диспетчера IFS, выдавая специальный системный вызов монтирования

(для тома) или регистрации (для сетевой файловой системы). В процессе регистрации FSD передает Диспетчеру IFS адреса своих входных точек. Эти входные точки однозначно соответствуют файловым системным вызовам из состава API. Таким образом, в самой ОС почти совсем отсутствуют общая обработка системных вызовов и основная функция IFS – опознание того FSD, которому следует переадресовать вызов. Опознание адресата Диспетчер IFS производит следующим образом:

- если одним из параметров вызова является полное имя файла, Диспетчер IFS использует идентификатор диска для определения соответствующего FSD по таблице томов;
- если одним из параметров вызова является манипулятор открытого файла, Диспетчер IFS использует манипулятор для определения соответствующего FSD по таблице открытых файлов;
- если происходит запрос, с которым Диспетчер IFS не может разобраться, он поочередно вызывает все зарегистрированные FSD, пока один из них не примет запрос.

Подсистема ввода-вывода – IOS – выполняет две функции: управляющего модуля, который распределяет обращения, и сервисные функции для драйверов.

Драйвер отслеживания тома – VTD – контролирует смену носителей на устройствах.

Специфический дисковый драйвер – TSD – управляет всеми устройствами определенного типа и преобразует виртуальные адреса на устройстве в физические.

Драйверы независимых разработчиков – VSD – (необязательные) выполняют любую дополнительную обработку данных.

Драйвер порта – аппаратно-зависимый драйвер, выполняющий обмен данных с устройством на низком уровне.

В Windows 95 идея независимой от драйверов модификации данных реализована таким образом, что в ОС определено общее количество звеньев в цепочке обработки, которую проходят данные, – 32. IOS составляет уровень 0, драйвер порта – уровень 31. Между ними, следовательно, возможно еще 30 уровней, которые могут быть заняты драйверами VSD, при установке драйвера VSD можно выбрать уровень, на котором он будет включен в цепочку.

Что касается конкретной файловой системы, то файловые системы Windows 95 – VFAT (виртуальная FAT) и FAT-32. Их структура в принципе такая же, как и FAT-16, но VFAT позволяет работать с длинными именами файлов (за счет того, что одному файлу может соответствовать несколько элементов каталога: основной и продолжения), а в FAT-32 вдобавок к этому элемент FAT имеет размер 32 бита, следовательно, позволяет работать с большим числом кластеров на диске. Структура элемента каталога – та же, что и у FAT-16, но размер имени файла может достигать 250 символов. Это достигается тем, что для файлов с длинными именами выделяется несколько элементов каталога, и специальная комбинация атрибутов файла отличает элемент, содержащий продолжение имени от первого элемента. Кроме того, корневой каталог в FAT-32 уже не привязан жестко к определенному месту на диске, следовательно, снимается ограничение на его размер.

3.5. Другие свойства Windows 9x

Средства взаимодействия процессов в Windows 95 – семафоры, события и переменные взаимного исключения. Каждое из этих средств имеет собственный API для создания и управления, но для ожидания на всех объектах синхронизации используются одни и те же системные вызовы.

Интерфейс Windows 95 хорошо известен всем пользователям, так что нет нужды описывать его подробно. Он создавался по образцу объектно-ориентированного интерфейса Mac OS фирмы Apple. Однако, интерфейс Windows 95 является не объектно-, а документо-ориентированным. Объектами в строгом смысле являются файлы программ и данных. Некоторые управляющие компоненты (в том числе и сам Рабочий Стол) объектами не являются, а только имитируют объектное поведение. В Windows 98 в связи с интеграцией в нее Web-браузера внешний вид интерфейса несколько изменился, но принципы его построения и функционирования остались те же.

Поскольку Windows 95 является однопользовательской системой, задача защиты ресурсов ставится в ней только в смысле разделения ресурсов разных процессов. Принятые в системе "облегченные" методы такого разделения создают потенциальную возможность для возникновения ошибок в системе, для устранения которых потребовалась бы тщательная отладка системного кода. Любому пользователю Windows 95, однако, хорошо известно, что такая отладка фирмой Microsoft не производится. Но эту ситуацию можно если не простить, то понять. Windows 95 является персональной ОС и не предназначена для выполнения критических приложений. Как правило, сбой системы не приводит к потерям большим, чем несколько минут, потраченных на перезагрузку, и испорченное настроение. Пока пользователи продолжают покупать ОС, пусть даже и ненадежно работающую, для фирмы нет смысла увеличивать затраты на повышение надежности.

С самого начала фирма Microsoft рассматривала Windows 95 как некий суррогат, призванный обеспечить удержание фирмой позиций до выхода в свет новой ОС (сначала предполагалось, что это будет Windows NT v.4, затем – Windows 2000). Ориентация на широкий круг непрофессиональных пользователей и активная, хотя и не всегда честная

рыночная политика привели к тому, что Windows 95 стала монополистом на рынке персональных ОС. Версия Windows 98 строится на том же ядре и отличается прежде всего интегрированным в систему Web-браузером Microsoft Internet Explorer. В 2000 году появилась новая версия – Windows ME (Millenium Edition). Следует ожидать, что эта версия будет не только использоваться, но и развиваться в течение еще нескольких лет прежде, чем уступит место более надежным системам (например, Windows 2000).

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Чем отличается многозадачность Windows 9x от многозадачности Windows 3.x?
2. К чему приводит использование части системных модулей Windows 3.x в Windows 9x?
3. Почему утверждается, что в Windows 9x встроена MS DOS?
4. Какую из описанных в части I моделей памяти реализует Windows 9x?
5. Каким из способов, описанных в главе 9 части I, реализованы в Windows 9x общие области памяти?
6. Какие недостатки файловой системы FAT-16 устранены в FAT-32? Какие недостатки унаследованы?
7. Каким из способов, описанных в главе 7 части I, реализована файловая система в Windows 9x?
8. Ответьте честно и мотивированно: если бы вы были в руководстве фирмы Microsoft, стали ли бы вы добиваться стабильной работы Windows 9x?

Глава 4. Операционная система OS/2

4.1. История и архитектура

Первая версия OS/2 появилась в 1987 г. и являлась совместной разработкой фирм IBM и Microsoft. В ходе работы над следующими версиями фирма Microsoft, во-первых, сочла завышенными требования IBM к надежности, во-вторых, решила делать ставку на свой продукт Windows и прекратила свое участие в проекте.

Первая версия OS/2 предназначалась для компьютеров на базе процессора Intel 80286 с его 16-разрядным словом и сегментной моделью виртуальной памяти. В последующих версиях и релизах поэтапно вводились: новая файловая система, графический интерфейс, сегментно-страничная модель памяти, 32-разрядность. Однако, все релизы первой и второй версий OS/2 [6] предъявляли требования к ресурсам, превышающие средний уровень имеющихся в то время ПЭВМ, поэтому эти версии были несколько "тяжеловесны". Этот недостаток был устранен в версии 3 – OS/2 Warp (1995 г.), в которой все свойства ОС были оптимизированы в компактном ядре. Эта версия стала выходить как клиентской, так и в серверной редакции (в последнюю был включен продукт IBM LAN Server).

Архитектура OS/2 Warp [20] показана на рисунке 4.1.

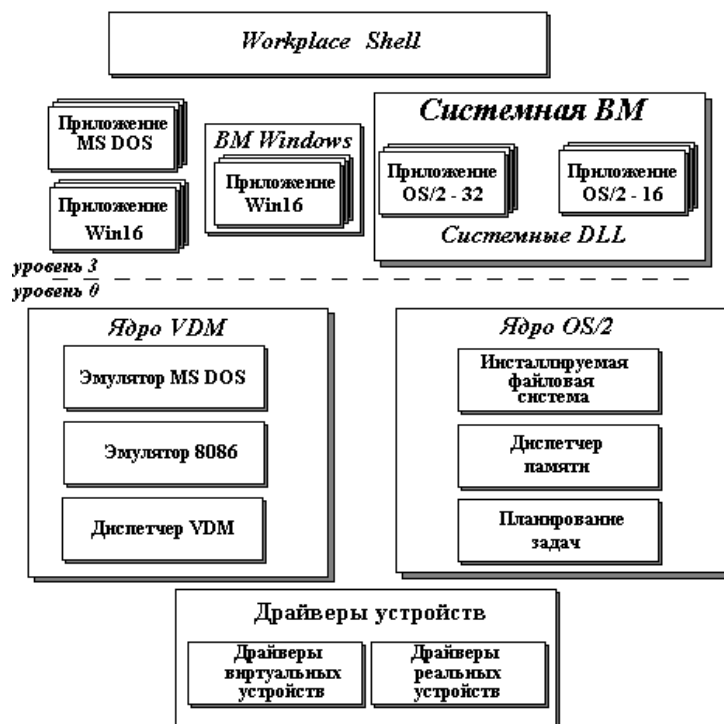


Рисунок 4.1 Архитектура OS/2 Warp

При значительном сходстве с архитектурой Windows 9x архитектура OS/2 Warp обладает рядом существенных отличий от нее.

Системная виртуальная машина OS/2 обеспечивает выполнение приложений OS/2 – 16- и 32-разрядных. Отдельная виртуальная машина создается для эмуляции среды Windows 3.x, в этой среде 16-разрядные приложения Windows выполняются в общем адресном пространстве, в режиме невытесняющей многозадачности – как и в Windows 9x. Однако, можно запускать приложения Windows и в отдельных адресных пространствах, тогда они выполняются в режиме вытесняющей многозадачности вместе с другими приложениями Windows, OS/2 и MS DOS. Приложения MS DOS выполняются каждое в среде собственной виртуальной машины MS DOS. Отдельной частью ядра OS/2 является ядро VDM (виртуальной машины MS DOS), которое обеспечивает эмуляцию функций MS DOS (в отличие от Windows 9x здесь нет кода ОС MS DOS),

эмуляцию процессора Intel 8086 и диспетчеризацию виртуальных машин MS DOS.

Графическая оболочка Workplace Shell является отдельным приложением, запускаемым опционно, OS/2 может функционировать и без графической оболочки, в режиме командной строки.

4.2. Многозадачность

С самого начала OS/2 являлась многозадачной системой с вытесняющей многозадачностью. Многозадачность в этой системе имеет три уровня: сеанс, процесс, нить.

Сеанс – это окно на экране. Сеанс может быть запущен в полноэкранном или оконном режиме. Каждый сеанс имеет собственную виртуальную консоль, включающую в себя логические эквиваленты монитора, клавиатуры и мыши; командный процессор (CMD.COM в сеансах OS/2 или COMMAND.COM в сеансах MS DOS); начальный командный файл (OS2ININ.CMD в сеансах OS/2 или AUTOEXEC.BAT в сеансах MS DOS). Сеанс предоставляет пользователю самостоятельную рабочую среду (MS DOS, Windows 3.x или OS/2). Когда сеанс выдвигается на передний план, соответствующая сеансу виртуальная консоль становится эквивалентной физической консоли. API OS/2 позволяет порождать новые сеансы и управлять из родительского сеанса состоянием сеанса дочернего.

В каждом сеансе может быть запущен один или (только в сеансе OS/2) несколько процессов. В соответствии с общепринятым подходом процессу в OS/2 соответствует программа с набором выделенных ей ресурсов. В API системы имеются системные вызовы для порождения нового процесса с выполнением в нем другой программы или для смены программы, выполняемой в текущем процессе. Порождаемые процессы связаны с породившим отношениями "потомок–предок". Наследование

ресурсов (файлов, каналов) может устанавливаться для каждого экземпляра ресурса избирательно.

Каждый процесс состоит из одной или нескольких нитей. Нить является объектом планирования процессорного времени. Нити разделяют большую часть ресурсов процесса, в составе которого они выполняются, но каждая нить имеет собственный контекст и собственный стек.

С самого начала OS/2 проектировалась как система с вытесняющей многозадачностью. Участки нереентерабельного кода в ядре системы минимизированы, а MS DOS и Windows также эмулируются ядром, поэтому OS/2 в состоянии обеспечить более оперативное переключение процессов, чем Windows 95. OS/2 управляет процессами в режиме квантования времени, размер кванта является параметром, задаваемым при загрузке системы. При освобождении процессора или по истечении кванта активным назначается процесс с наивысшим приоритетом. В системе имеются следующие 4 класса приоритетов процессов (в порядке убывания приоритетности):

- критический – для процессов реального времени и сетевых коммуникаций; для процессов этого класса гарантируется время реакции не более 6 мксек;
- серверный – для процессов, выполняющих запросы от других процессов;
- нормальный – для интерактивных процессов;
- отложенный – для процессов, работающих без доступа к терминалу.

Внутри каждого класса приоритет процесса перевычисляется динамически (имеется еще по 32 градации приоритета в каждом классе) по таким правилам:

- процесс, окно которого является в данный момент активным, получает "добавку переднего плана";
- процесс, выполняющий операцию ввода-вывода, получает "добавку ввода-вывода", которая делает его приоритет наивысшим в классе; по окончании операции ввода-вывода эта добавка отбирается у процесса;
- процесс, пребывающий в состоянии ожидания дольше некоторого времени (задаваемого при загрузке) получает "добавку голодания", которая ставит его сразу после критического класса и позволяет практически немедленно получить квант процессорного времени, после использования кванта эта добавка отбирается у процесса.

4.3. Управление памятью

Ранние версии OS/2 были ориентированы на сегментную модель виртуальной памяти. OS/2 Warp обеспечивает для процесса плоскую модель памяти, хотя поддерживает также и API старой, сегментной модели. Структура виртуального адресного пространства процесса OS/2 показана на рисунке 4.2.

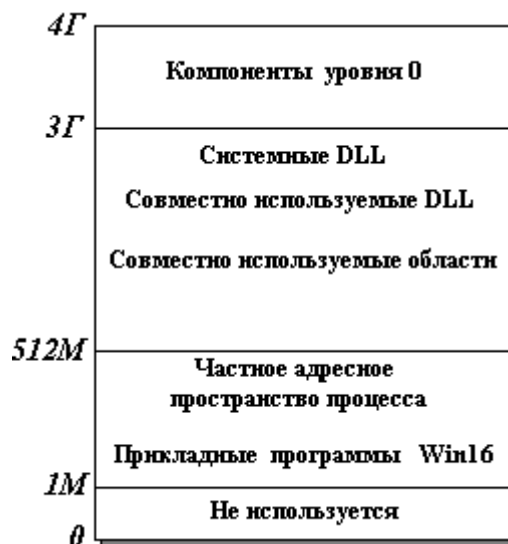


Рисунок 4.2 Адресное пространство процесса OS/2

Качественно структура адресного пространства процесса – такая же, как и в Windows 95. В последних релизах OS/2 Warp Server for e-business граница частного адресного пространства процесса может быть поднята до 3 Гбайт. Плоская модель памяти обеспечивается теми же средствами, что и в Windows 95: единственный сегмент для процесса и использование каталога страниц. Однако, в OS/2 на одни и те же таблицы страниц второго уровня указывают только элементы страничных каталогов, относящихся к адресному пространству выше 3 Гбайт. Поэтому объекты, совместно используемые несколькими процессами, имеют разные виртуальные адреса для разных процессов.

Динамическая компоновка во время загрузки является важным компонентом системы, и большинство системных модулей оформлены в виде библиотек динамической компоновки – DLL. Имеется также возможность компоновки во время выполнения – в этом случае требуемый модуль DLL должен быть явным образом загружен программой и определены адреса его входных точек, для чего имеется соответствующий API.

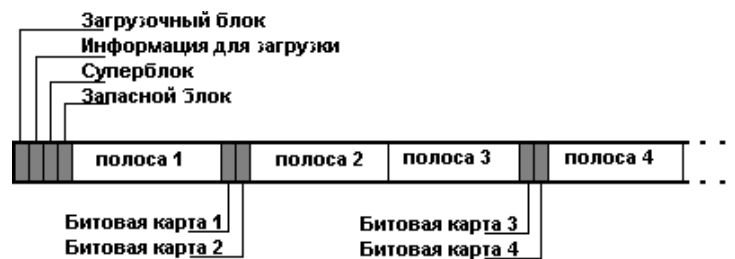
4.4. Устройства и файловая система

Драйверы устройств имеют "классическую" двухуровневую структуру и устанавливаются при загрузке системы. Драйверы выполняются на уровне защиты 2 процессора Intel/Pentium, что дает им возможность выполнять команды ввода-вывода, но не другие привилегированные команды. Выполнение низкоуровневых системных функций (например, управление реальной памятью) обеспечивается для драйверов системным сервисом – внутренним вызовом `DosHep1`.

Подобно Windows 95, OS/2 обеспечивает устанавлируемые файловые системы. Только файловая система FAT-16 поддерживается ядром ОС. Сетевая файловая система и CDFS поддерживаются через механизм устанавлируемой файловой системы. Основная же устанавлируемая файловая система OS/2 – HPFS.

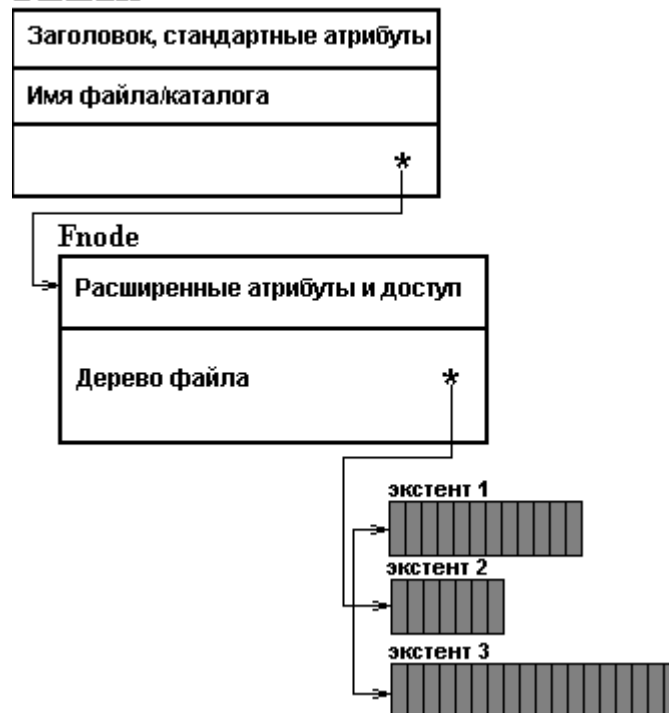
HPFS – High Performance File System (высокопроизводительная файловая система) – разработана совместно фирмами IBM и Microsoft в период их совместной работы над OS/2 версии 1.2. HPFS была призвана прежде всего заменить FAT MS DOS и отличается от последней высокой эффективностью в управлении жесткими дисками большого объема и поддержкой длинных (до 255 символов имен файлов). Структура тома в HPFS показана на рисунке 13.10.а. Начальная часть диска (16 блоков) резервируется под блок загрузки и загрузочную информацию, суперблок, запасной блок (копию суперблока) и т.д. Корневой каталог HPFS расположен в логической середине диска, чтобы минимизировать перемещение головок. Эти системные структуры используются для загрузки ОС, поддержки работы файловой системы и восстановления. Распределяемое дисковое пространство разбивается на так называемые полосы, размером по 8 Мбайт каждая. Каждой полосе соответствует битовая карта размером 2 Кбайт. Один элемент битовой карты

соответствует одному 512-байтному блоку (сектору) полосы и отражает его состояние (занят/свободен). Битовые карты поочередно размещаются в начале и в конце каждой полосы, таким образом, две смежные полосы образуют непрерывный участок дискового пространства размером в 16 Мбайт. Файловая система планирует размещение новых файлов на диске по возможности в непрерывном участке дискового пространства. Кроме того, за концом нового файла оставляется свободный участок "на вырост", что дает файлу возможность в будущем расширяться, не теряя непрерывности в размещении.



а). структура тома HPFS

DIRBLK



б). элемент каталога HPFS

Рисунок 4.3 Файловая система HPFS

Организация файлов и каталогов в HPFS включает в себя две структуры, показанные на рисунке 4.3.б: элемент каталога (DIRBLK) и дескриптор файла или каталога (Fnode). В суперблоке содержится указатель на дескриптор корневого каталога. Каталог состоит из элементов каталога, которые организованы в сбалансированное двоичное дерево, упорядоченное для поиска по имени файла/каталога. В элементе каталога содержится указатель на дескриптор файла/каталога. Основное содержание дескриптора – план размещения файла. Как отмечалось выше, файловая система стремится разместить весь файл в одном экстенте, но не гарантирует такого размещения. Файлы, характеризующиеся значительной изменчивостью, могут занимать большое число экстентов. Элемент плана размещения состоит из начального адреса экстента и его длины. Описания 8 экстентов могут быть размещены непосредственно в дескрипторе. Если же файл состоит из большего числа экстентов, план его размещения структурируется в В-дерево с корневым узлом в дескрипторе.

Для повышения производительности HPFS использует кеширование данных при записи на диск и отложенную запись

Новая версия этой файловой системы – HPFS386 оптимизирована для работы с новыми поколениями процессоров Intel/Pentium и большими дисковыми системами. Она отличается тем, что значительная часть драйвера файловой системы работает на уровне защиты 0. В ней значительно увеличены размеры кешей, сняты некоторые ограничения, повышена надежность (поддерживается технология RAID-1). HPFS386 интегрируется с IBM LAN Server и обеспечивает хранение списков контроля доступа в файловой системе.

4.5. Средства взаимодействия

OS/2 обладает полным набором средств взаимодействия процессов, описанным нами в главе 9 части 1.

В системе имеется 9 типов сигналов (в том числе 3 – пользовательских) с возможностью устанавливать их собственную обработку или игнорировать сигнал (кроме сигнала KILL).

Общие области памяти выделяются как сегменты и могут быть именованными или неименованными.

Неименованные каналы создаются специальным системным вызовом (одним вызовом создается один канал – для чтения или для записи), дальнейшая работа с каналом происходит с использованием API файлов. Для именованных каналов наряду с файловым API имеются специальные операции, обеспечивающие обмен данными в одном вызове (открытие канала, запись, чтение, закрытие).

Очереди сообщений обеспечиваются с применением общих областей памяти, в которых размещаются тела сообщений. Собственно очередь в системной области составляют только заголовки сообщений, а область памяти, в которой находится тело сообщения, становится доступной процессу-получателю через манипулятор, содержащийся в заголовке.

В системе различаются 3 типа семафоров: семафоры событий, служащие для синхронизации нитей и процессов, семафоры взаимного исключения, служащие для защиты ресурсов от одновременного доступа, и семафоры множественного ожидания, обеспечивающие ожидание на нескольких семафорах событий или семафорах взаимного исключения одновременно. Для каждого типа семафоров имеется свой набор системных вызовов, обеспечивающих создание/уничтожение семафоров и выполнение семафорных операций.

Все именованные средства взаимодействия вписываются в пространство имен файловой системы, и имена их выглядят как имена файлов, расположенных в специальных каталогах.

4.6. Другие свойства OS/2

Хотя OS/2 в настоящее время позиционируется на рынке как серверная ОС, ее ядро продолжает оставаться однопользовательским, то есть на уровне ядра OS/2 пользователей не различает. Защита в ядре относится прежде всего к защите ресурсов процессов, которая обеспечивается надежной изоляцией адресных пространств процессов друг от друга. Контроль доступа обеспечивается промежуточным программным обеспечением (IBM LAN Server и др.), поставляемым "в одной коробке" с OS/2.

Графический интерфейс OS/2 во многом напоминает интерфейс Windows 95. Это объясняется тем, что разработчики обеих систем брали за образец интерфейс MacOS. Однако IBM разрабатывала интерфейс OS/2 в непосредственном сотрудничестве с фирмой Apple, поэтому он не только внешне подобен образцу, но и полностью воплощает его объектно-ориентированные свойства. Workplace Shell (рабочий стол) и Warp Center (панель быстрого доступа, подобная линейке программ в Windows 9x) являются приложениями, запускаемыми по выбору.

OS/2 может также работать и в режиме командной строки. Набор команд OS/2 является расширением набора команд MS DOS. Наиболее интересной из этого расширения нам представляется команда CALL, которая позволяет запустить программу без ожидания ее завершения. Именно команда CALL создает возможность запускать несколько процессов в одном сеансе OS/2. OS/2 имеет богатейшие возможности для командных файлов, которые обеспечиваются языком REXX, являющимся

неотъемлемой частью ОС. Команды и программы, выполняемые в интерпретаторе REXX, имеют возможность обмениваться сигналами и данными, используя перенаправление ввода-вывода или интерфейсы очередей.

В первые два года своего существования OS/2 Warp, а затем и ее версия 4 – Merlin – конкурировала на рынке персональных ОС с Windows 95. Так OS/2 Merlin стала первой ОС со встроенной поддержкой мультимедиа, голосового ввода и Java-платформой. Несмотря на то, что по объективным показателям OS/2, по крайней мере, не уступала своему конкуренту, она потерпела поражение, прежде всего – из-за отсутствия должной рекламы. В 1998 г. фирма IBM решила, что рынок персональных систем не входит в сферу ее стратегических интересов, и позиционировала OS/2 как серверную систему. В таком качестве OS/2 приобрела значительное число корпоративных пользователей, особенно в Европе. Версия 5 OS/2 – Aurora, расширенная прежде всего поддержкой SMP-архитектуры и файловой системой JFS, заимствованной из ОС AIX (см. главу 7), появилась в 1999 г. Долгое время она существовала только в серверном варианте, и только в 2001 г. появилась клиентская редакция этой версии. Хотя OS/2 продолжает эксплуатироваться и развиваться, в настоящее время фирма IBM исключила из своих стратегических интересов любые ОС для платформы Intel/Pentium (а возможно – и саму эту платформу) и не занимается продвижением OS/2 на рынке. Поскольку стратегия разработки приложений IBM в последнее время диктует ориентацию на платформенную независимость, IBM предлагает использовать OS/2 прежде всего как платформу для разработчиков Java-приложений с последующим переносом результатов на другие платформы. Предложение вполне разумное, так как OS/2 является, во-первых, чрезвычайно устойчивой, а во-вторых, виртуальная машина Java от IBM

для OS/2 является одной из самых эффективных (если не самой эффективной) из всех существующих.

Вместе с тем, в движении Открытых Кодов существует стойкое и активное ядро сторонников OS/2, не желающих отказываться от нее даже как от настольной системы. Среди акций этого движения можно назвать давно и успешно развивающийся проект по обеспечению выполнения среде OS/2 приложений Win32, а также недавно начавшееся движение за перевод OS/2 в открытые коды. Можно предполагать, что эта OS/2 не прекратит своего существования даже, если IBM от нее совсем откажется.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как изменялась модель памяти OS/2 в ходе ее эволюции?
2. Каковы уровни многозадачности в OS/2?
3. Встроена ли MS DOS в OS/2?
4. Каким из способов, описанных в главе 9 части I, реализованы в OS/2 Warp общие области памяти? Сопоставьте это с их реализацией в Windows 9x.
5. В чем преимущества файловой системы HPFS по сравнению с FAT-16? по сравнению с FAT-32?
6. Чем принципиально отличается Workplace Shell OS/2 Warp Desktop Windows 9x?
7. OS/2 Warp позиционируется на рынке как серверная ОС. Почему же мы считаем возможным рассматривать ее как персональную ОС?
8. Считаете ли вы, что как персональная ОС OS/2 Warp лучше, чем Windows 9x? Если да, то почему, по-вашему, она проиграла в конкурентной борьбе?

Глава 5. Операционные системы Windows NT/2000

5.1. История и архитектура

Windows NT явилась продолжением фирмой Microsoft проекта OS/2, предпринятым фирмой Microsoft после того, как разошлись ее пути с IBM. В качестве руководителя проекта Windows NT был приглашен Д.Катлер, имеющий большой опыт в разработке операционных систем в фирме DEC (ОС VAX VMS).

С самого начала Windows NT планировалась как ОС, предназначенная для выполнения функций сервера. Windows NT является полностью 32-разрядной ОС с объектно-ориентированной структурой и строится на базе микроядра. Последнее обстоятельство позволило сделать ОС доступной на большом числе аппаратных платформ CISC- и RISC-процессоров, в том числе и в симметричных многопроцессорных архитектурах. Архитектура ОС [10, 35] представлена на рисунке 5.1.

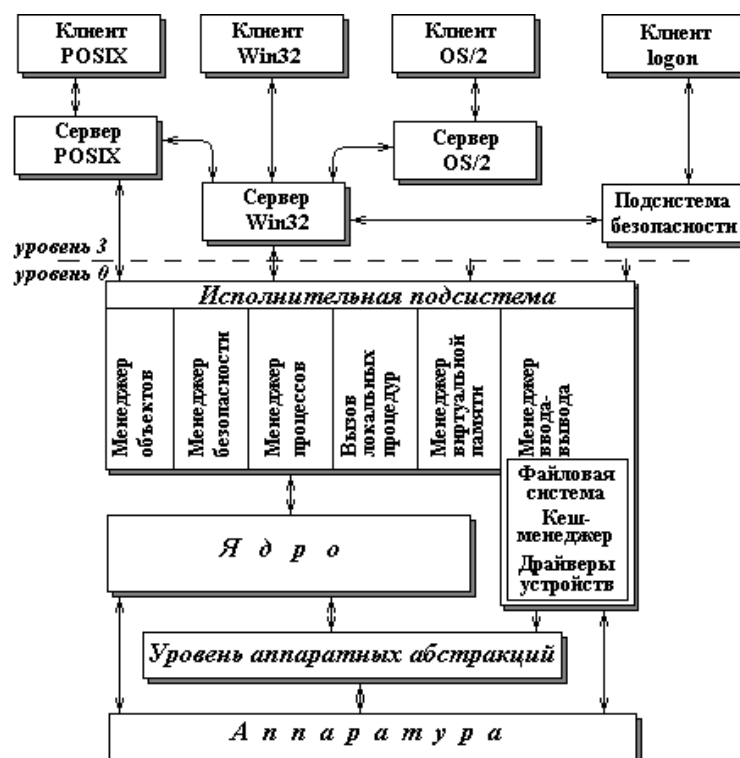


Рисунок 5.1 Архитектура Windows NT/2000

Реализация концепции микроядра в Windows NT состоит в том, что ОС состоит из процессов-серверов, выполняющих непосредственное обслуживание пользовательских процессов в пользовательском режиме, и части системы, работающей в режиме ядра, выполняющей по запросам процессов-серверов низкоуровневые и критические операции.

Часть системы, работающая в режиме ядра, состоит из нескольких слоев.

Уровень аппаратных абстракций виртуализирует аппаратные интерфейсы. При его создании ставилась цель подготовки процедур, которые позволяли бы единственному драйверу конкретного устройства поддерживать функционирование этого устройства на всех аппаратных платформах. Аппаратные абстракции скрывают от вышележащих частей ОС такие детали как интерфейс ввода-вывода, контроллеры прерываний и другие аппаратно-зависимые и специфические функции. Уровень аппаратных абстракций также обеспечивает программные механизмы управления и обмена данными между процессорами в симметричных

многопроцессорных архитектурах. Ядро и драйверы устройств для выполнения аппаратно зависимых функций обращаются к уровню аппаратных абстракций, хотя начиная с версии 4.0, для повышения быстродействия допускается непосредственное взаимодействие драйвера с аппаратурой. Модуль уровня аппаратных абстракций требует перенастройки при изменении конфигурации оборудования.

5.2. Ядро и планирование процессов

Ядро осуществляет планирование действий процессора и синхронизацию работы процессов и нитей. Ядро является резидентным и непрерываемым. Ядро объектно-базировано, то есть обеспечивает низкоуровневую базу для определенных объектов ОС, которые могут использоваться компонентами высшего уровня. Объекты ядра делятся на две группы: объекты управления и объекты диспетчеризации. Основным объектом управления является процесс, представляющий собой адресное пространство, набор доступных процессу объектов и совокупность нитей управления. Некоторые другие объекты управления: прерывание, процедура синхронного вызова, процедура отложенного вызова и т.д. Объекты диспетчеризации характеризуются сигнальными состояниями и управляют диспетчеризацией и синхронизацией операций. Примеры объектов диспетчеризации: нить, семафор, событие, взаимное исключение (mutex – для пользовательского режима и mutant – для режима ядра) и другие.

Ядро реализует основную политику планирования процессов и нитей (хотя в нее могут быть внесены изменения серверами подсистем). Всего в Windows NT имеется 32 градации приоритетов, разнесенные по 4 классам. При запуске процесс получает уровень приоритета, назначаемый по умолчанию его классу:

- для класса реального времени – уровень 24;
- для высокого класса – уровень 13;
- для нормального класса – уровень 9 для интерактивного или уровень 7 для фонового процесса;
- для отложенного класса – уровень 4.

Общие правила диспетчеризации таковы:

- на выполнение выбирается нить с наивысшим уровнем приоритета, ей выделяется квант процессорного времени;
- если появляется нить с более высоким уровнем приоритета, выполнение текущей нити прерывается, текущая нить ставится в очередь готовых, при этом ее приоритет не изменяется;
- если нить исчерпала выделенный ей квант времени, она прерывается и ставится в очередь готовых, при этом ее приоритет понижается;
- если нить перешла в ожидание, не исчерпав выделенный ей квант времени, ее приоритет повышается.

Исполнительная подсистема – верхний уровень ядра, представляющий сервис ядра подсистемам среды и другим серверам. Ниже перечисляются компоненты исполнительной подсистемы.

Диспетчер объектов обеспечивает поддержку объектно-базированной структуры ОС, представляющей ресурсы в виде объектов – абстрактных инкапсулированных типов данных. Диспетчер Объектов выполняет общие функции управления объектами – как пользовательскими, так и объектами ОС. Все пользовательские объекты имеют внешние имена. Объекты ОС могут быть как именованными, так и неименованными, последние используются только внутри самой ОС. Именование любых объектов подчиняется общим правилам, имена объектов подобны именам файлов в иерархической структуре

каталогов. Диспетчер объектов обеспечивает унифицированные правила для именования, хранения и обеспечения безопасности объектов.

Менеджер безопасности проводит политику защиты и аудита объектов на персональном компьютере (подробнее – см.ниже).

Менеджер процессов отслеживает объекты процессов и нитей. Фактически основная часть планирования процессов и нитей реализована в ядре, менеджер же процессов обеспечивает создание, уничтожение и протоколирование этих объектов, а также обеспечивает набор средств создания и использования нитей и процессов для конкретных подсистем сред. В Windows NT процессы не связываются родственными отношениями в структуру дерева (как, например, в Unix или в OS/2). Однако в Windows 2000 введен новый объект – задание, представляющий собой именованную группу процессов и позволяющий вести общее управление и учет для такой группы.

5.3. Адресные пространства

Менеджер виртуальной памяти выполняет формирование виртуального адресного пространства процесса и отображает виртуальные адреса в адресных пространствах процессов на физические страницы памяти. Структура виртуального адресного пространства пользовательского процесса показана на рисунке 5.2.

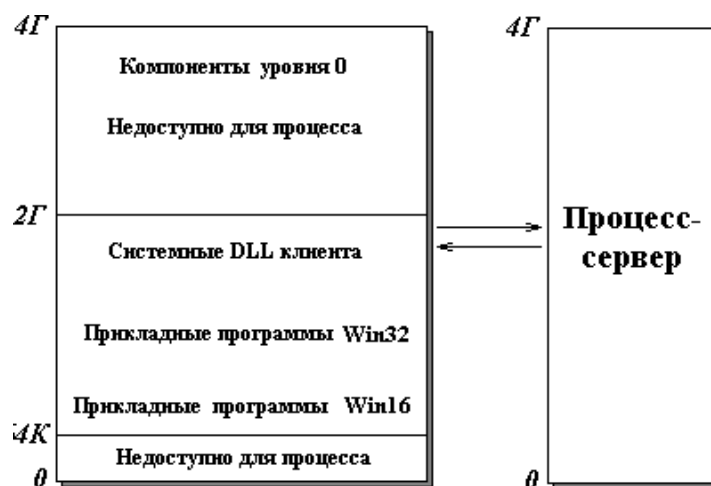


Рисунок 5.2 Виртуальное адресное пространство пользовательского процесса в Windows NT

Из 4 Гбайт виртуального адресного пространства пользовательского процесса доступна нижняя половина за вычетом самых младших 64 Кбайт. В виртуальном адресном пространстве процесса реализована плоская модель памяти, качественно структура доступной для процесса памяти совпадает с таковой в Windows 9x, но в Windows NT адресные пространства процессов полностью изолированы друг от друга. Верхняя часть виртуального адресного пространства процесса, в которой находятся системные DLL, в Windows NT содержит не сами эти DLL, а только модули-заглушки. Обращение процесса к системе происходит в пределах адресного пространства процесса. Но такое обращение попадает к модулю-заглушке, который формирует сообщение-запрос к подсистеме-серверу на выполнение системного вызова. Средства **вызова локальных процедур** передают это сообщение процессу-серверу, он же передает ответ сервера в модуль-заглушку, а тот формирует отклик на системный вызов. У пользовательского процесса, таким образом, создается впечатление, что системный вызов был выполнен в пределах его адресного пространства, но если пользовательский процесс испортит верхнюю часть доступного ему

адресного пространства, то он испортит только свои модули-заглушки и никак не повлияет на работу других процессов.

5.4. Ввод-вывод

Менеджер ввода-вывода обеспечивает независимый от устройств интерфейс ввода-вывода и отвечает за пересылку запросов на ввод-вывод соответствующим драйверам. Менеджер Ввода-Вывода поддерживает драйверы файловых систем, драйверы устройств и сетевые драйверы, обеспечивая для них однородную среду. Модель ввода-вывода Windows NT многоуровневая, каждый драйвер отвечает за логически законченный уровень работы. Самый нижний уровень составляют драйверы устройств. Другие драйверы являются надстройкой над драйверами устройств и не зависят от специфики работы конкретного устройства. Поддержка файловой системы отдельным драйвером в этой иерархии позволяет Windows NT работать с разными файловыми системами.

5.5. Процессы-серверы

Часть системы, работающая в пользовательском режиме, представлена процессами-серверами. Процессы-серверы выполняются каждый в своем адресном пространстве, полностью изолированном от пользовательских процессов и от других серверов. Серверу доступны все 4 Гбайта его виртуального адресного пространства. Процессы-серверы могут быть подсистемами сред или специальными серверами. Подсистемы сред обеспечивают для пользовательских процессов среды выполнения, соответствующие спецификациям тех или иных операционных систем. Подсистема работает в пользовательском режиме, но ее виртуальное

адресное пространство полностью отделено от адресного пространства любого приложения. Взаимодействие между приложениями и подсистемами происходит только через вызовы локальных процедур, что делает процесс-сервер защищенным от клиентов. Основной и обязательной подсистемой является Win32, остальные среды могут устанавливаться по выбору. Подсистема Win32, которая обеспечивает выполнение приложений Windows NT и Windows 9x, а также эмулирует среды MS DOS и Windows 3.1. Среда Win32 обеспечивает наиболее полную функциональность, и другие серверы обращаются к ней для выполнения некоторых функций. Приложения MS DOS и Windows 3.1 выполняются в контексте процесса, являющегося Виртуальной Машиной MS DOS. Виртуальная Машина MS DOS – это процесс Win32, который эмулирует процессор Intel 8086, прерывания BIOS, прерывания MS DOS и системные функции Windows 3.1 и драйверы реального режима. Каждая Виртуальная Машина MS DOS выполняется в собственном изолированном виртуальном адресном пространстве и обеспечивает для выполняющихся в ней приложений адресное пространство, соответствующее MS DOS или Windows 3.1.

Специальными серверами являются службы Windows NT, такие как регистратор событий, подсистема безопасности, средства вызова удаленных процедур и т.п., а также компоненты промежуточного программного обеспечения фирмы Microsoft, устанавливаемые по выбору – такие как MS SQL Server, MS Transaction Server и т.п. Со временем значимость подсистем сред (кроме среды Win32) падает, так как Microsoft не ставит перед собой задачи поддержки приложений, разработанных для других сред, но значительно возрастает роль серверных процессов, в которых выполняются продукты семейства MS BackOffice.

5.6. Система безопасности

Windows NT с самого начала планировалась как многопользовательская система, поэтому средства аутентификации пользователей и авторизации доступа к ресурсам встроены в систему, в том числе и на уровне ядра. На рисунке 5.3. показаны компоненты системы безопасности Windows NT (имеется в виду локальная безопасность, т.е. защита ресурсов на локальном компьютере).

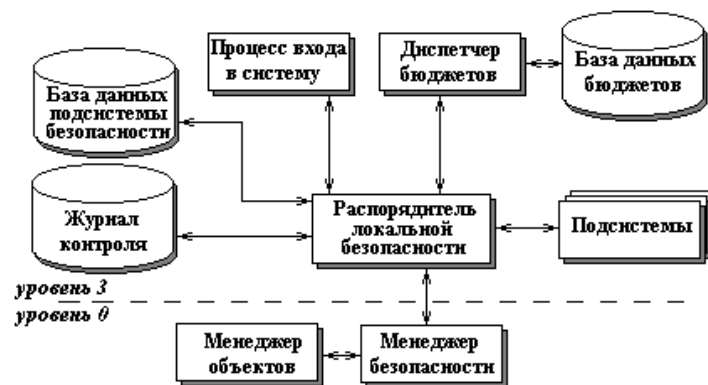


Рисунок 5.3 Система безопасности Windows NT

Процесс входа в систему обеспечивает начало работы с системой как интерактивных пользователей, так и удаленных клиентов, обращающихся к серверу. Этот компонент не связан с безопасностью непосредственно, но обращается к Распорядителю локальной безопасности, который организует аутентификацию пользователя по имени и паролю. Бюджеты всех зарегистрированных пользователей сохраняются в Базе данных бюджетов, доступ к которой осуществляется только через Диспетчер бюджета безопасности. Кроме того, Распорядитель локальной безопасности управляет политикой локальной безопасности – разрешениями на доступ и ведет Базу данных локальной безопасности, а

также управляет политикой контроля и регистрирует контрольные сообщения в Журнале безопасности.

Основную работу выполняет Менеджер безопасности в составе Исполнительной системы. Этот модуль проверяет права доступа к объектам по запросам других модулей Исполнительной системы (прежде всего – Менеджера объектов) и генерирует контрольные сообщения. Для получения информации о правах и передачи контрольных сообщений Менеджер безопасности взаимодействует с Распорядителем локальной безопасности.

При входе пользователя в систему для него на основе информации о бюджете пользователя, хранящейся в Базе данных бюджетов, создается маркер доступа, содержащий:

- уникальный идентификатор безопасности пользователя;
- список идентификаторов безопасности групп, в которые пользователь входит;
- специальные привилегии пользователя.

Последний элемент определяет права, не связанные с отдельными объектами, например: выполнять низкоуровневую отладку объектов, увеличивать приоритет процессов, изменять конфигурацию драйверов, работать с Журналом контроля и т.д. Пользователи могут входить в группы, в Windows NT существует несколько встроенных групп (например, Администраторы, Операторы сервера, операторы архива и т.д.), которым некоторые специальные привилегии предоставляются по умолчанию.

Копия маркера доступа создается для каждого процесса данного пользователя. Однако, для работы в режиме клиент/сервер Windows NT использует так называемое воплощение прав. Суть воплощения состоит в том, что процесс-сервер (или его нить), выполняющий обслуживание клиента, выполняется с маркером доступа процесса-клиента.

Windows NT использует модель безопасности, основанную на списках контроля доступа: основная информация о возможностях доступа связывается с объектом, а не с пользователем. Все именованные и некоторые неименованные объекты имеют собственные дескрипторы безопасности в Базе данных локальной безопасности. В дескриптор безопасности входят:

- идентификатор владельца объекта;
- идентификаторов группы владельца (используется только для подсистемы POSIX);
- контролируемый список управления доступом, определяющий, какие пользователи имеют доступ к объекту, этот список управляется владельцем объекта;
- системный список управления доступом, определяющий, какие контрольные сообщения должны генерироваться для этого объекта, этот список управляется Администратором.

Контролируемый список управления доступом состоит из элементов двух типов: "доступ разрешается" и "доступ запрещается". Каждый из элементов содержит идентификатор пользователя и маску доступа, кодирующую одну или несколько возможностей для данного пользователя. Для одного пользователя может быть несколько элементов в списке, описывающих его различные возможности, при просмотре списка эти возможности складываются.

Каждый тип объекта может иметь до 5 специфических возможностей – операций, выполняемых только для данного типа объекта. Кроме того, имеются стандартные возможности – операции, выполняемые для объектов любого типа:

- синхронизация доступа к объекту;
- назначение владельца;

- запись;
- чтение;
- удаление.

При задании прав доступа возможно задание так называемых общих возможностей, которые представляют собой макросы, кодирующиеся в списке управления доступом определенными комбинациями стандартных и специфических возможностей.

Авторизация, то есть проверка правильности доступа к объекту состоит из таких шагов.

1. При обращении пользователя к объекту Менеджер безопасности создает запрашиваемую маску доступа, в ней кодируются те возможности, которые заданы в запросе.
2. Менеджер безопасности поэлементно просматривает контролируемый список управления доступом. Если в списке находится элемент, содержащий идентификатор того пользователя, который выдал запрос, элемент обрабатывается.
3. Если элемент имеет тип "доступ запрещается", то при совпадении хотя бы одной возможности в маске доступа элемента с возможностью в запрашиваемой маске доступа, доступ отклоняется. Все элементы типа "доступ запрещается" располагаются в начале списка, следовательно, обрабатываются в первую очередь.
4. Если доступ отклонен, то проверяется, не содержит ли запрашиваемая маска только возможности чтения или записи и не является пользователь владельцем объекта. Если выполняются эти условия, то доступ предоставляется.
5. Если элемент имеет тип "доступ разрешается", то ищутся совпадения возможностей в маске доступа элемента с возможностями в запрашиваемой маске доступа. Если достигнуто

согласование, доступ предоставляется, в противном случае просмотр списка продолжается.

6. Если при достижении конца списка согласование не достигнуто, доступ отклоняется.

Системный список управления доступом состоит из элементов только одного типа. Элемент этого списка содержит идентификатор пользователя, маску контролируемых возможностей и флаги, определяющие генерацию сообщения при успешном или неуспешном доступе. Обработка системного списка управления доступом похожа на обработку контролируемого списка. Если установлено соответствие между запрашиваемым доступом и маской в элементе списка, то в зависимости от значения флагов в элементе и результатов проверки прав по контролируемому списку может быть сгенерировано сообщение о событии.

Windows NT предоставляет также возможность отслеживания событий, относящихся ко всей системе в целом. Различаются несколько категорий таких событий:

- изменения в Базе данных бюджетов;
- изменения в Базе данных локальной безопасности;
- события отслеживания процессов;
- вход в систему и выход из нее;
- использование привилегий;
- доступ к защищенному объекту;
- системные события.

Перечень контролируемых событий – как системных, так и событий, связанных с отдельными объектами – конфигурируется Администратором.

5.7. Файловая система NTFS

Механизм устанавливаемой файловой системы позволяет Windows NT работать с различными файловыми системами. Однако наиболее высокоэффективную работу вычислительных систем с несколькими жесткими дисками большого объема обеспечивает файловая система NTFS. Кроме того, NTFS поддерживает длинные имена файлов и обеспечивает надежное хранение и повышенные возможности для восстановления информации. NTFS в полной мере использует все средства контроля доступа свойственные Windows NT как объектно-ориентированной системе.

В соответствии с общей объектно-ориентированной архитектурой Windows NT каждый файл в NTFS представляется как объект, имеющий определенный набор атрибутов. К атрибутам относятся, например: имя файла, тип файла, временные отметки, дескриптор безопасности, индексная информация (для каталогов) и т.д. Данные файла также являются его атрибутом. Каждый файл в виде списка своих атрибутов представлен записью в специальном файле, называемом Главной Файловой Таблицей (MFT – Master File Table). Размер записи в MFT не фиксирован, но ограничен сверху (до 2 Кбайт). Примерная структура записи MFT показана на рисунке 5.4.

Заголовок	Стандартная информация	Имя файла или каталога	Дескриптор безопасности	Данные или указатели на данные	...
-----------	------------------------	------------------------	-------------------------	--------------------------------	-----

Рисунок 5.4 Структура записи MFT в файловой системе NTFS

Атрибуты файла, размещаемые непосредственно в записи MFT называются резидентными. Так, имя файла – обязательно резидентный атрибут. Атрибуты, не поместившиеся в записи MFT, становятся нерезидентными, они размещаются в отдельных экстентах, а запись MFT содержит указатели на них.

В зависимости от размера файла возможны следующие варианты его размещения (в порядке возрастания размера):

- данные файла целиком размещаются в области данных записи MFT;
- область данных записи MFT содержит список адресов и размеров экстентов, занимаемых файлом;
- вместо области данных в записи MFT содержатся расширенные атрибуты, в которых имеется указатель на другую запись MFT, которая содержит только заголовок и список адресов и размеров экстентов, занимаемых файлом;
- расширенные атрибуты содержат список указателей на другие записи MFT, каждая из которых содержит заголовок и список экстентов файла.

Подобным же, хотя и несколько отличным образом размещаются и каталоги:

- небольшой каталог представляет собой список имен файлов, который целиком размещается в области данных MFT;
- для большого каталога список имен файлов в области данных MFT также имеется, однако добавляется еще атрибут, называемый индексом размещения, содержащий указатели на другие записи MFT; список имен файлов совместно с индексом размещения образуют корень В-дерева, упорядоченного таким образом, чтобы минимизировать время поиска файла по некоторому атрибуту (как

правило, этим атрибутом является имя файла, но возможно упорядочение по любому резидентному атрибуту).

Несколько первых записей MFT зарезервированы для системных нужд. Первая запись описывает сам файл MFT, вторая – его резервную копию. Адрес MFT и копии записаны в секторе начальной загрузки и продублированы в логическом центре диска. Вся системная информация представлена в виде файлов – записей MFT – хотя и скрытых от просмотра. Системными файлами являются:

- сама MFT;
- резервная копия MFT;
- файл регистрации транзакций;
- дескриптор тома;
- корневой каталог;
- информация начальной загрузки;
- битовый массив кластеров;
- список плохих кластеров.

Дисковая память в NTFS распределяется кластерами, размер кластера может выбираться.

Надежность NTFS обеспечивается прежде всего регистрацией транзакций. NTFS использует "ленивую" запись – занесение в кэш информации о любой транзакции. Параллельно, как фоновый процесс, эта информация записывается в файл регистрации транзакций. Каждые несколько секунд NTFS проверяет кэш, чтобы определить состояние отложенной записи и фиксирует это состояние в файле регистрации как контрольную точку. При возникновении сбоя файловая система приводит свое состояние к последней контрольной точке, после чего обрабатывает записи регистрации транзакций, сделанные после фиксации контрольной точки: повторно выполняет все завершенные транзакции и отменяет

(откатывает) незавершенные. Такая система гарантирует сохранение целостности тома, то есть соответствие метаданных файловой системы ее действительному состоянию.

Для обеспечения надежности в отношении пользовательских данных в Windows NT имеется возможность программной поддержки технологий RAID. К таким средствам относятся:

- RAID 0 – зеркальное копирование дисков – поддержка двух идентичных разделов на разных дисковых дорожках, управляемых одним контроллером;
- RAID 1 – дублирование дисков – поддержка двух идентичных разделов на дисковых дорожках с разными контроллерами;
- RAID 5 – чередование дисков с контролем четности – технология, обеспечиваемая программно на уровне драйвера, поддерживается только в Windows NT Server, в Workstation обеспечивается только чередование, без контроля.

NTFS совместима с файловыми системами FAT и HPFS. Для согласования с POSIX в NTFS включаются дополнительные средства, используемые в подсистеме выполнения POSIX-приложений.

Следует признать, что концепции, положенные в основу архитектуры Windows NT, являются изящными и хорошо продуманными и способны обеспечить как высокую функциональность и эффективность ОС, так и ее безопасность. Справедливые претензии пользователей к надежности и к уязвимости системы для хакерских атак объясняются, по мнению специалистов (см., например [4]), ошибками и небрежностью в реализации в основном системных и сетевых служб, но отчасти – и ядра системы. В старших версиях ОС разработчики с целью повышения быстродействия допускают нарушение некоторых концептуальных свойств системы (например, взаимодействия драйверов с оборудованием только через Уровень аппаратных абстракций, взаимодействия клиентов с сервером

только через Вызов локальных процедур и т.п.), что снижает надежность системы.

Поскольку на момент создания Windows NT основные интересы фирмы Microsoft были сосредоточены на персональной сфере применения компьютеров первая версия этой ОС (v.3.1) не получила должного развития и поддержки. Начиная с версии 3.5, Windows NT попадает в число стратегических продуктов фирмы Microsoft, широкое распространение получает версия 4, а версия 5, переименованная в Windows 2000, объявляется Microsoft единственным стратегическим продуктом в сфере ОС. Предполагалось, что Windows 2000 будет существовать в различных редакциях – от "тонкого" клиента до промышленного сервера, и, по замыслу Microsoft, станет единственной операционной системой фирмы. Однако, на сегодняшний день (середина 2001 г.) на рынке всерьез рассматриваются только ее серверные редакции. Персональные редакции по объему потребляемых ресурсов пока не могут конкурировать с ОС семейства Windows 9х, хотя отличаются от них значительно большей надежностью. Большинство производителей промышленного промежуточного и прикладного программного обеспечения (например, IBM, Oracle, Inprise) объявили о "готовности своих продуктов к Windows 2000" практически в день ее появления в продаже. Но они не рекомендуют своим покупателям спешить с миграцией на нее с Windows NT 4, предлагая подождать, пока в Windows 2000 не будут исправлены основные ошибки. Однако, следует ожидать, что в ближайшие 2-3 года такая миграция произойдет повсеместно, Windows 2000 как серверная ОС продолжит конкуренцию с клонами Unix.

Судя по последней (начало 2002 г.) "смене вывески", фирма Microsoft намерена разделить все свои ОС на два семейства продуктов – Windows XP и Windows .NET Server. Оба семейства должны базироваться на ядре Windows 2000. Windows XP масштабируется от версии Embedded до

Server. Windows .NET Server представляет серверы промышленного масштаба, обеспечивающие концепцию Microsoft .NET. Концепция .NET представляет собой архитектуру взаимодействия в глобальном информационном пространстве, на основе которой фирма Microsoft намерена вступить борьбу с промышленными стандартами интероперабельности и с фирмами, обеспечивающими взаимодействие в глобальной сети (Sun Microsystems, IBM и др.).

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Является ли Windows NT/2000 ОС на базе микроядра?
2. Какие тенденции архитектуры ОС реализованы в Windows NT/2000?
3. Назовите основные компоненты Windows NT/2000 и их функции.
4. Как выполняется планирование процессов в Windows NT/2000?
5. Какова структура адресного пространства процесса в Windows NT/2000?
6. Почему можно считать, что память в Windows NT/2000 защищена значительно лучше, чем в Windows 9x или в OS/2?
7. Что такое процессы-серверы в Windows NT/2000? Какие существуют типы процессов-серверов?
8. Как взаимодействуют компоненты Windows NT/2000 в системе безопасности?
9. Опишите сценарий процесса авторизации в Windows NT/2000.
10. В чем преимущества файловой системы NTFS по сравнению с FAT? по сравнению с HPFS?

Глава 6. Семейство ОС Unix

6.1. История и современное состояние

Писать об ОС Unix чрезвычайно трудно. Во-первых, потому, что об этой системе написано очень много. Во-вторых, потому, что идеи и решения Unix оказали и оказывают огромное влияние на развитие всех современных ОС, и многие из этих идей уже описаны в этой книге. В-третьих, потому что Unix – не одна ОС, а целое семейство систем, и не всегда можно "отследить" их родство между собой, а уж описать все ОС, входящие в это семейство просто невозможно. Тем не менее, мы, ни в коей мере не претендуя на полноту, попытаемся дать беглый обзор "мира Unix" в тех в тех его областях, которые представляются нам интересными для целей нашего учебного курса.

Рождение ОС Unix относится к концу 60-х годов, и эта история уже обросла "легендами", которые подчас по-разному повествуют о деталях этого события. ОС Unix родилась в исследовательском центре Bell Telephone Laboratories (Bell Labs), входящей в состав корпорации AT&T. Изначально этот инициативный проект для ЭВМ PDP-7 (впоследствии – для PDP-11) представлял собой то ли с файловую систему, то ли компьютерную игру, то ли систему подготовки текстов, то ли и то, и другое, и третье. Важно, однако, то, что с самого начала проект, превратившийся в итоге в ОС, задумывался как программная среда коллективного пользования. Автором первой версии Unix является Кен Томпсон, однако, в обсуждении проекта, а впоследствии – и в его реализации принимал участие большой коллектив сотрудников (Д.Ритчи, Б.Керниган, Р.Пайк и другие). На наш взгляд, несколько счастливых обстоятельств рождения Unix определили удачу этой системы на много лет вперед.

Для большинства сотрудников того коллектива, в котором родилась ОС Unix, эта ОС была "третьей системой". Существует мнение (см., например [1]), что системный программист достигает высокой квалификации только при выполнении третьего своего проекта: первый проект получается еще "ученическим", во второй разработчик пытается включить все, что не получилось в первом и в итоге он получается слишком громоздким, и только в третьем достигается необходимый баланс желаний и возможностей. Известно, что до рождения Unix коллектив Bell Labs участвовал (совместно с рядом других фирм) в разработке ОС MULTICS. Конечный продукт MULTICS (Bell Labs не принимала участия в последних стадиях разработки) носит все признаки "второй системы" и не получил широкого распространения. Следует, однако, заметить, что в этом проекте были рождены многие принципиально важные идеи и решения, и некоторые концепции, которые многие считают рожденными в Unix, на самом деле имеют своим источником проект MULTICS.

ОС Unix была системой, которая делалась "для себя и для своих друзей". Перед Unix не ставилась задача захвата рынка и конкуренции с какими-либо продуктами. Сами разработчики ОС Unix были и ее пользователями, и сами оценивали соответствие системы своим нуждам. Без давления рыночной конъюнктуры такая оценка могла быть предельно объективной.

ОС Unix явилась системой, которая сделана программистами и для программистов. Это определило изящество и концептуальную стройность системы – с одной стороны, а с другой – необходимость понимания системы для пользователя Unix и чувства профессиональной ответственности для программиста, разрабатывающего программное обеспечение для Unix. И никакие последующие попытки сделать "Unix для чайников" не смогли избавить ОС Unix от этого достоинства.

В 1972-73 гг. Кен Томпсон и Деннис Ритчи написали новую версию Unix. Специально для этой цели Д.Ритчи создал язык программирования C, представлять который теперь уже нет необходимости. Более 90% программного кода Unix написано на этом языке, и язык стал неотъемлемой частью ОС. То, что основная часть ОС написана на языке высокого уровня, обеспечивает возможность ее перекомпиляции в коды любой аппаратной платформы и является обстоятельством, определившим широкое распространение Unix.

В период создания Unix антимонопольное законодательство США не давало корпорации AT&T возможности выходить на рынок программных продуктов. Поэтому ОС Unix была некоммерческой и свободно распространялась, прежде всего – в университетах. Там ее развитие продолжалось и наиболее активно оно велось в Калифорнийском университете в Беркли. При этом университете была создана группа Berkeley Software Distribution, которая занималась развитием отдельной ветви ОС – BSD Unix. На протяжении всей последующей истории основная ветвь Unix и BSD Unix развивались параллельно, неоднократно взаимно обогащая друг друга.

По мере распространения ОС Unix стал все более возрастать интерес к ней коммерческих фирм, которые стали выпускать собственные коммерческие версии этой ОС. Со временем стала коммерческой и "основная" ветвь Unix от AT&T, для ее продвижения была создана дочерняя фирма Unix System Laboratory. Ветвь BSD Unix в свою очередь разветвилась на коммерческую BSD и Free BSD [18]. Различные коммерческие и свободно распространяемые Unix-подобные системы строились на базе ядра AT&T Unix, однако в них включались и свойства, заимствуемые из BSD Unix, а также и оригинальные свойства. Несмотря на общий источник, различия между членами семейства Unix накапливались и в итоге привели к тому, что перенос приложений из одной Unix-

подобной ОС в другую стал чрезвычайно затруднен. По инициативе пользователей Unix возникло движение за стандартизацию API Unix. Это движение было поддержано Международной организацией стандартов ISO и привело к возникновению стандарта POSIX (Portable Operation System Interface eXecution), который развивается и в настоящее время и является самым авторитетным стандартом для ОС. Однако, оформление спецификаций POSIX как официального стандарта – процесс довольно медленный, и он не может удовлетворять потребностей производителей программного обеспечения, что привело к возникновению альтернативных промышленных стандартов.

С переходом AT&T Unix к компании Nowell название этой ОС изменилось на Unixware, а права на торговую марку Unix перешли к консорциуму X/Open. Этот консорциум (в настоящее время – Open Group) разработал свои (более широкие, чем POSIX) спецификации системы, известные как Single Unix Specification. Недавно вышла вторая редакция этого стандарта, значительно лучше согласованная с POSIX.

Наконец, ряд фирм – производителей собственных версий Unix образовал консорциум Open Software Foundation (OSF), который выпустил собственную версию Unix – OSF/1, сделанную на базе микроядра Mach. OSF также выпустил спецификации системы OSF/1, на основе которой фирмы-члены OSF стали выпускать собственные Unix-системы. Среди таких систем: SunOS фирмы Sun Microsystems, AIX фирмы IBM, HP/UX фирмы Hewlett-Packard DIGITAL UNIX фирмы Compaq и другие.

Поначалу Unix-системы этих фирм в большей степени базировались на BSD Unix, но сейчас большая часть современных промышленных Unix-систем в большей степени подобны AT&T Unix System V Release 4 (S5R4), хотя наследуют и некоторые свойства BSD Unix. Мы не берем на себя ответственность сравнивать коммерческие Unix-системы, так как

периодически появляющиеся в печати сравнения такого рода зачастую представляют совершенно противоположные результаты.

Компания Nowell продала Unix компании Santa Crouse Operations, которая выпускала собственный Unix-продукт SCO Open Server. SCO Open Server [16] базировался на более ранней версии ядра (System V Release 3), но был великолепно отлажен и отличался высокой стабильностью. Фирма Santa Crouse Operations интегрировала свой продукт с AT&T Unix и выпустила Open Unix 8 [30], однако затем продала Unix фирме Caldera, которая и является владельцем "классической" ОС Unix сегодня (в конце 2001 г.).

Фирма Sun Microsystems начала свое представительство в мире Unix системой SunOS, созданной на основе ядра BSD. Однако впоследствии заменила ее системой Solaris на основе S5R4 [34]. В настоящее время распространяется версия 8 этой ОС (существует также v.9-бета). Solaris работает на платформе SPARC (RISC-процессоры, изготавливаемые по спецификациям Sun) и Intel-Pentium.

Фирма Hewlett-Packard предлагает ОС HP-UX.v.11 на платформе PA-RISC [22]. HP-UX базируется на S5R4, но содержит много свойств, "выдающих" ее происхождение от BSD Unix. Конечно же, HP-UX будет доступна и на платформе Intel-Itanium.

Фирма IBM выступает с ОС AIX, последняя на сегодняшний день версия – 5L (о ней еще пойдет речь впереди) [12]. IBM не объявляла "родословную" AIX, это в основном оригинальная разработка, но первые версии носили признаки происхождения от FreeBSD Unix. Сейчас, однако, AIX больше похожа на S5R4. Первоначально ОС AIX была доступна и на платформе Intel-Pentium, но впоследствии (в соответствии с общей политикой IBM) перестала поддерживаться на этой платформе. В настоящее время AIX работает на серверах IBM RS/6000 и в других

вычислительных платформах на базе процессоров PowerPC (в том числе и на суперкомпьютерах IBM).

ОС DIGITAL UNIX фирмы DEC была единственной промышленной реализацией системы OSF/1. ОС DIGITAL UNIX работала на RISC-серверах Alpha фирмы DEC. Когда в 1998 г. фирма DEC была поглощена фирмой Compaq, в фирму Compaq перешли и серверы Alpha, и DIGITAL UNIX. Фирма Compaq имеет намерение восстановить присутствие на рынке серверов Alpha и в связи с этим интенсивно развивает и ОС для них. Нынешнее название этой ОС – Tru64 Unix (текущая версия – 5.1A), она продолжает базироваться на ядре OSF/1 и несет в себе много признаков BSD Unix [37].

Несмотря на то, что большинство коммерческих Unix-систем базируется на одном ядре и удовлетворяет требованиям POSIX, каждая из них имеет собственный диалект API, и различия между диалектами накапливаются. Это приводит к тому, что перенос промышленных приложений с одной Unix-системы на другую затрудняется и требует, как минимум, перекомпиляции, а часто – и корректировки исходного кода. Попытка преодолеть "разброд" и сделать единую для всех ОС Unix была предпринята в 1998 г. альянсом фирм SCO, IBM и Sequent. Эти фирмы объединились в проекте Monterey с целью создания единой ОС на базе Unixware, владельцем которой в то время была SCO, IBM AIX и ОС DYNIX фирмы Sequent. (Фирма Sequent занимает лидирующие позиции в производстве ЭВМ архитектуры NUMA – несимметричной многопроцессорной – и DYNIX – это Unix для таких ЭВМ). ОС Monterey должна была работать на 32-разрядной платформе Intel-Pentium, 64-разрядной платформе PowerPC и на новой 64-разрядной платформе Intel-Itanium. О поддержке проекта заявили почти все лидеры производства аппаратных средств и промежуточного программного обеспечения. Даже фирмы, имеющие собственные клоны Unix (кроме Sun Microsystems),

объявили, что на платформах Intel они будут поддерживать только Monterey. Работа над проектом продвигалась, по-видимому, успешно. ОС Monterey была в числе первых, доказавших свою работоспособность на Intel-Itanium (наряду с Windows NT и Linux) и единственной, которая при этом не прибегала к эмуляции 32-разрядной архитектуры Intel-Pentium. Однако в финальной стадии проекта произошло фатальное событие: SCO продала свое Unix-отделение. Еще раньше фирма Sequent вошла в состав IBM. "Наследником" всех свойств ОС Monterey стала ОС IBM AIX v.5L. Однако, не совсем всех. Платформа Intel-Pentium не является для IBM стратегическим направлением, и на этой платформе ОС AIX недоступна. А поскольку другие лидеры компьютерной индустрии не разделяют (или не вполне разделяют) такую позицию IBM, идея общей ОС Unix так и не реализовалась.

Наконец, нельзя не сказать и о некоммерческих ОС, которые в той или иной степени могут считаться относящимися к семейству Unix. О FreeBSD мы уже упоминали. Это, по-видимому, лучший из некоммерческих продуктов, и эта ОС уже давно нашла себе применение в промышленной обработке данных.

Последние годы отмечены шумной экспансией ОС Linux [9, 26]. Ядро этой ОС было разработано в 1991 Линусом Торвальдсом (Финляндия) прежде всего "для личного пользователя". Торвальдс сделал исходный код своей ОС открытым. При фиксированном ядре любой программист может написать собственные сервисы ОС Linux и опубликовать их через Internet. Трудно сказать, что послужило причиной такой популярности ОС Linux. Она не является ни уникальной, ни лучшей ни как свободно распространяемая, ни как открытая, ни как Unix для компьютеров небольшой вычислительной мощности. По-видимому, этот феномен объясняется некоторым совпадением объективных и субъективных факторов, анализ которых не входит в наши цели.

До 1997 г. ОС Linux была популярна почти исключительно в университетской среде. Однако в 1997 г. ряд фирм объявил о выпуске собственных версий Linux. Некоторые из этих версий (например, Red Hat) остались бесплатными, некоторые (например, Caldera) стали коммерческими. Принципиально важно то, что у Linux появились "хозяева", которые несли ответственность (в том числе и коммерческую) за сопровождение ОС. Это вызвало стремительное возрастание интереса к Linux пользователей, которые решают задачи промышленной обработки информации. Такой интерес объясняется, с одной стороны, желанием иметь "почти настоящую" ОС Unix со значительно меньшими расходами, с другой, нежеланием попадать в полную зависимость от Microsoft. Фирмы-лидеры информационных технологий не могли игнорировать настроения рынка и объявили либо о выпуске собственных версий Linux (например, Hewlett-Packard), либо о поддержке определенных версий Linux на своих аппаратных платформах и в своем промежуточном программном обеспечении (например, IBM). Интересно, что эти фирмы являются производителем собственных коммерческих версий ОС Unix. В последних версиях всех коммерческих ОС Unix, рассмотренных выше, в API ОС внесены системные вызовы, обеспечивающие возможность выполнения в них приложений, написанных для Linux. Пока, по-видимому, гранды информационных технологий не видят в Linux серьезного конкурента для своих ОС в сфере промышленной обработки данных и систем высокой готовности и предусматривают перенос информационных систем с Linux на коммерческие Unix при достижении ими определенной степени зрелости. Не без помощи "грандов" достигнут перенос Linux на большое число платформ – от встроенных вычислительных устройств до мейнфреймов и суперкомпьютеров. Возможно, с достижением Linux промышленного уровня, в этой ОС сможет воплотиться идея, не удавшаяся в проекте Monterey – единая ОС Unix для всех.

Следует отметить еще одну интересную сферу применения Linux. Ряд стран (например, Китай, Россия) объявил о принятии в качестве базовой ОС для информационных системах в своих силовых структурах ОС, "изготавливаемой на основе Linux". Основную роль здесь играет открытость исходного кода. Доступность исходного кода Linux позволяет правительственным специалистам этих стран с одной стороны убедиться в отсутствии в ОС "закладок", занимающихся "электронным шпионажем", а с другой – провести исчерпывающую верификацию ОС и добиться ее высокой надежности и безопасности.

В следующем изложении мы ориентируемся прежде всего на "классическую" ОС Unix, ведущую свою родословную от Bell Labs, и ее последнюю на сегодняшний день версию – Open Unix 8, но рассматриваем также и некоторые особенности других коммерческих и некоммерческих ОС семейства Unix.

6.2. Архитектура Unix

Архитектура ОС Unix [14] представлена на рисунке 6.1.

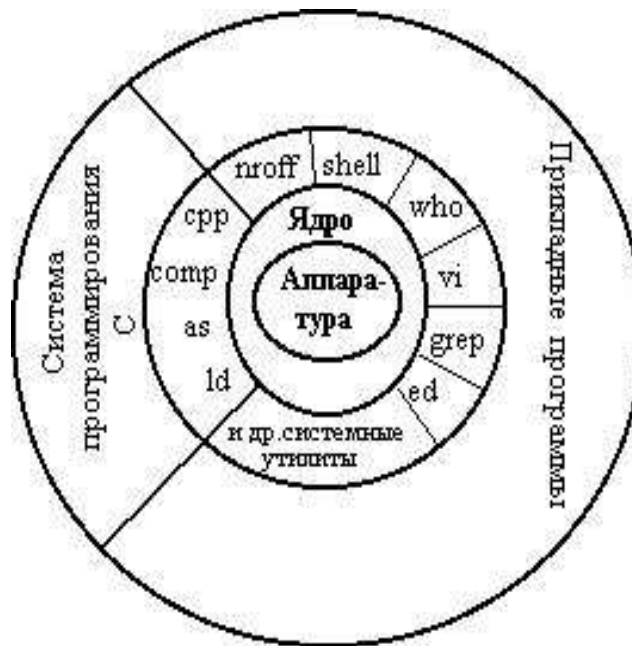


Рисунок 6.1 Архитектура Unix

Хотя эта архитектура объявлена как иерархическая, в ней просматриваются всего три уровня иерархии: ядро, системные утилиты, процессы пользователей. Само ядро Unix, таким образом, является монолитным, хотя некоторая (не формулируемая декларативно) структурированность в нем, конечно же, имеется. Так, в Unix различаются две принципиально различные группы системных вызовов: внутренние, которые доступны для пользовательских процессов и составляют собственно API ОС, и внешние, употребляемые только из модулей ядра. Весьма тщательно выполнено разделение ядра на аппаратно-зависимую и аппаратно-независимую части.

Модули первой части пишутся на языке Ассемблера целевой платформы и выполняют:

- загрузку и инициализацию ОС;
- переключение контекста;
- управление реальной памятью;
- первичную (низкоуровневую) обработку прерываний;

- низкоуровневое управление устройствами (аппаратные драйверы).

Модули второй части написаны на языке С и обеспечивают высокоуровневую обработку, в том числе:

- порождение планирование и прочее управление процессами и (в поздних реализациях) нитями;
- управление виртуальной памятью;
- средства взаимодействия между процессами;
- поддержку файловой системы;
- логическую обработку прерываний;
- высокоуровневые функции драйверов устройств;
- поддержку API.

Обращения процессов к системе имеют вид программных прерываний, но из прикладной программы обращение к ОС имеет вид вызова библиотечных функций, написанных на языке С. Модули библиотеки системных вызовов аппаратно-зависимые и содержат обращения к ядру с использованием тех механизмов программного прерывания, которые доступны на данной аппаратной платформе.

6.3. Процессы

Концепция процесса в том виде, в каком она представлена в нашей книге, родилась именно в MULTICS и Unix. Процессом является выполняемая программа, которая представляет собой последовательности байтов, интерпретируемые процессором как машинные команды, данные и стековые структуры.

Контекстом процесса в Unix является его состояние, определяемое содержимым адресного пространства процесса, значениями глобальных переменных пользователя и информационными структурами, содержимым

машинных регистров, значениями, хранимыми в строке таблицы процессов, а также содержимым стеков задачи и ядра, данного процесса. Коды модулей ОС и общие информационные структуры, совместно используемые всеми процессами, не являются составной частью контекста процесса.

Вся работа системы происходит в контексте какого-либо процесса. Если процесс выполняет системный вызов, при котором происходит переход в режим ядра, то меняется режим, но не контекст. Модули ядра, следовательно, выполняются в контексте процесса и используют ресурсы того процесса, для которого они выполняются. Прерывания также выполняются в контексте процесса – того процесса, который был прерван, даже если прерывание не имеет отношения к этому процессу. Программы обработки прерываний обычно не работают со статической составляющей контекста процесса, так как эта часть не связана с прерываниями.

В системе процесс представляется строкой в таблице процессов. В ранних версиях Unix таблица процессов, как и другие системные структуры данных в памяти, располагались в статической памяти, имели фиксированный размер и смежное размещение. Позднее эти структуры стали располагаться в динамически выделяемой памяти со связным размещением, что дает возможность изменять их размер. Строка таблицы процессов содержит такую информацию о процессе, которая используется вне его контекста, такие как: идентификатор процесса, состояние процесса, его приоритет и т.д., а также указатели на структуры, описывающие контекст процесса.

Контекст процесса состоит из пользовательской и системной составляющих.

Пользовательский контекст включает в себя содержимое адресного пространства процесса (программного кода, данных, стека) и вектор состояния (содержимое аппаратных регистров).

Системный контекст содержит структуры данных ядра, связанные с этим процессом. Системный контекст процесса в Unix состоит из статической и динамических частей. Для каждого процесса имеется одна статическая часть контекста системного уровня и переменное число динамических частей.

Статическая часть системного контекста процесса включает в себя:

- строку таблицы процессов;
- часть адресного пространства задачи, выделенную процессу, где хранится управляющая информация о процессе; эта информация недоступна самому процессу, но может использоваться ядром только в контексте процесса;
- структуры данных, описывающие виртуальное адресное пространство процесса; если два или более процессов используют разделяемые сегменты памяти (см. ниже), то соответствующие структуры данных включаются в системный контекст каждого из этих процессов.

Динамическая часть системного контекста процесса представляет собой стек ядра.

Стек, входящий в состав пользовательской части контекста процесса предназначается для размещения в нем локальных переменных и обеспечения возвратов из вложенных вызовов процедур. Однако, как мы сказали выше, процесс может выполняться как в режиме процесса, так и в режиме ядра (точнее – модули ядра выполняются в контексте процесса). Динамическая часть системного процесса предназначена для обеспечения функционирования в режиме ядра. Выполнение модулей в режиме ядра может прерываться так же, как и выполнение модулей в режиме процесса. ОС Unix рассчитана на некоторую иерархию прерываний: каждое прерывание имеет некоторый уровень приоритета, и обработка прерывания какого-то уровня может быть прервано только прерыванием с

более высоким уровнем приоритета. Уровни приоритетов прерываний обычно поддерживаются аппаратной платформой. Типовое распределение уровней приоритетов (в порядке возрастания) следующее:

- программные прерывания,
- прерывания от терминалов,
- прерывания от сетевого оборудования,
- прерывания от дисков,
- прерывания от таймера,
- исключения (машинные сбои и исключительные ситуации).

Для каждого уровня прерывания создается свой стек ядра (или выделяется уровень в общем стеке ядра). В стеке i -го уровня сохраняется вектор состояния прерванного $i-1$ -го уровня (в стеке нижнего уровня – сохраняется вектор состояния прерванного процесса) и выделяется рабочее пространство стека для поддержки выполнения i -го уровня (например, для размещения его локальных переменных). Число стеков ядра (число динамических составляющих системного контекста) определяется количеством уровней прерываний, зависящим от платформы.

6.4. Нити

Нити появились в Unix довольно поздно, но они вошли в требования стандартов POSIX и Single Unix Specifications. Во всех современных клонах ОС Unix имеется механизм нитей, семантика (но не обязательно синтаксис) которого соответствует требованиям стандарта POSIX. В Open Unix 8 реализация нитей выполнена на уровне системной библиотеки нитей, однако потребовала и серьезной перестройки ядра. На уровне ядра работа с нитями поддерживается системными сущностями, называемыми легковесными процессами (light weight process). Нить представляет собой

отдельный поток управления с точки зрения программы. Легковесный процесс – отдельный поток управления с точки зрения ОС. Легковесный процесс предоставляет нити собственную среду выполнения, прежде всего – динамическую составляющую системного процесса, то есть стек ядра. Легковесные процессы могут интерпретироваться как виртуальные процессоры, как показано на рисунке 6.2. В каждый момент времени легковесный процесс может выполнять только одну нить, но нити могут сменять друг друга в легковесном процессе.

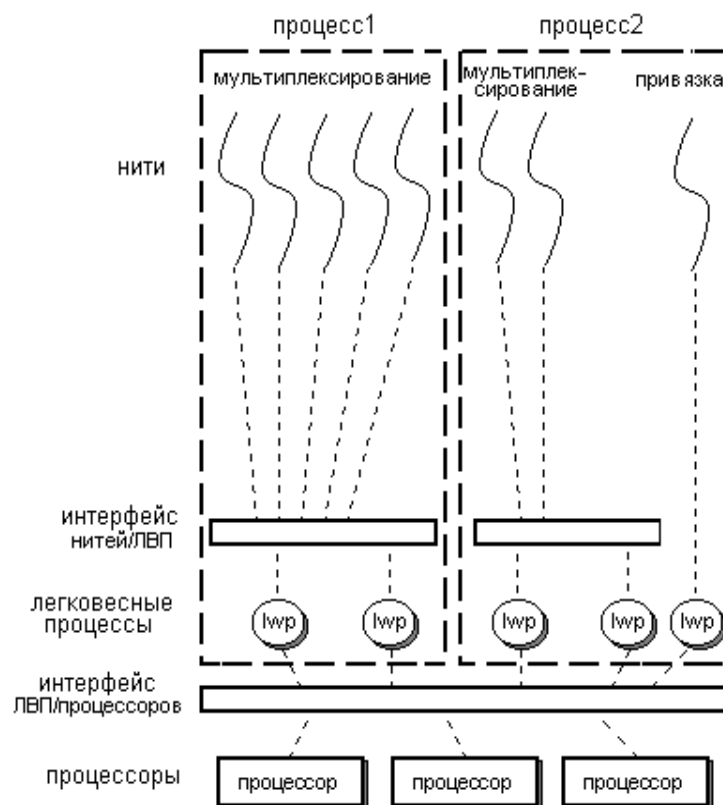


Рисунок 6.2 Нити и легковесные процессы (ЛВП)

Легковесные процессы подчиняются следующим правилам:

- с одним процессом может быть связано много легковесных процессов;

- каждый легковесный процесс разделяет адресное пространство "своего" процесса с другими легковесными процессами в том же процессе;
- каждый легковесный процесс разделяет легковесными процессами в том же процессе все ресурсы процесса (файлы, программные каналы и т.п.) и пользуется правами и привилегиями доступа своего процесса;
- каждый легковесный процесс является единицей планирования в ОС (ниже при описании дисциплины планирования мы везде под "процессом" имеем в виду "легковесный процесс");
- в мультипроцессорной системе несколько легковесных процессов одного процесса могут выполняться на разных процессорах одновременно (реальный параллелизм);

Нити могут быть мультиплексируемыми и связанными.

Системная библиотека нитей обеспечивает мультиплексирование нитей в пуле легковесных процессов каждого процесса по следующим правилам:

- легковесный процесс может выбирать и выполнять только одну нить за раз;
- по прошествии некоторого времени легковесный процесс оставляет текущую нить и выбирает другую;
- позже оставленная нить может быть выбрана вновь, не обязательно тем же легковесным процессом;
- в многопроцессорной системе имеется вероятность выполнения нитей с реальным параллелизмом.

Размер пула легковесных процессов называется реальным уровнем параллелизма. Уровень параллелизма устанавливается в конфигурационных параметрах, но может устанавливаться для процесса и

программно при помощи соответствующего системного вызова. Кроме того, он может изменяться динамически "вокруг" установленного уровня. Динамическое изменение уровня параллелизма выполняется библиотекой нитей для каждого процесса отдельно. При запуске каждого процесса для него создается только один легковесный процесс. Размер пула увеличивается, если создается новая нить с соответствующим параметром системного вызова `thr_create`. Если все легковесные процессы данного процесса оказываются в заблокированном состоянии, ядро посылает процессу специальный сигнал. По этому сигналу создается новый легковесный процесс. Как правило, количество легковесных процессов не превосходит количество нитей. Если легковесный процесс остается без работы в течение некоторого предустановленного интервала времени, он прекращается.

Связанные нити имеют собственный легковесный процесс для выполнения. Этот процесс создается одновременно с созданием нити (это определяется специальным параметром системного вызова `thr_create`) и уничтожается с уничтожением нити. Связанные нити не учитываются при определении реального уровня параллелизма. Связанная нить постоянно присоединена к своему легковесному процессу, когда такая нить переходит в ожидание, ее легковесный процесс также ожидает.

Мультиплексируемые же нити планируются на легковесные процессы – подобно тому, как процессы планируются на процессоры. Дисциплины планирования мультиплексируемых нитей реализуются библиотекой нитей, он "не видны" для ОС. Смена нитей на легковесном процессе происходит по событиям. Событиями, распознаваемыми библиотекой нитей, являются:

- освобождение легковесного процесса вследствие завершения выполнявшейся на нем нити, перехода нити в состояние ожидания

вследствие выполнения ею системного вызова или срабатывания механизмов синхронизации нитей (см. ниже);

- переход в состояние готовности нити с более высоким приоритетом, чем выполняющаяся;
- выполнение текущей нитью системного вызова `thr_yield`, которым она уступает свой легковесный процесс нити с таким же или более высоким приоритетом – если такие имеются.

Приоритеты нитей статические. Изначально созданная нить наследует приоритет нити, ее породившей, но приоритет может быть изменен явным образом при помощи системного вызова.

Приведенное выше описание механизма нитей мы строили на базе ОС Open Unix 8. Подобным же образом этот механизм реализован в ОС Solaris 8. В ОС AIX 5L "легковесные процессы" называются "нитеями ядра". В этих ОС также различаются связанные и мультиплексируемые нити.

Однако, ряд клонов реализуют совершенно иной механизм нитей. В ОС FreeBSD нити фактически являются отдельными процессами. В ОС FreeBSD имеется системный вызов `rfork`, который порождает новый процесс с возможностью управления наследованием ресурсов в новом процессе. Среди ресурсов, определяемых для наследования, может быть и адресное пространство процесса-родителя. Библиотечный вызов `rfork_thread` позволяет создать на базе вызова `rfork` нить и запустить в ней заданную процедуру.

Функционально аналогичный системный вызов – `clone()` – для порождения процессов-потомков с разделением адресного пространства и других (на выбор) ресурсов процесса-родителя существует в ОС Linux. Вызов `clone()` позволит задать флаги, указывающие, что порожденный процесс будет иметь со своим предком общие:

- адресное пространство;

- информацию о файловой системе: корневой и текущий каталоги;
- таблицу открытых файлов;
- таблицу обработчиков сигналов;
- родителя – в этом случае будет порожден не "дочерний", а "сестринский" процесс.

Нити в стандартной библиотеке поддержки нитей Linux реализованы просто как процессы, порожденные с указанием флага `CLONE_VM`, и с точки зрения ядра системы ничем не отличаются от любых других процессов.

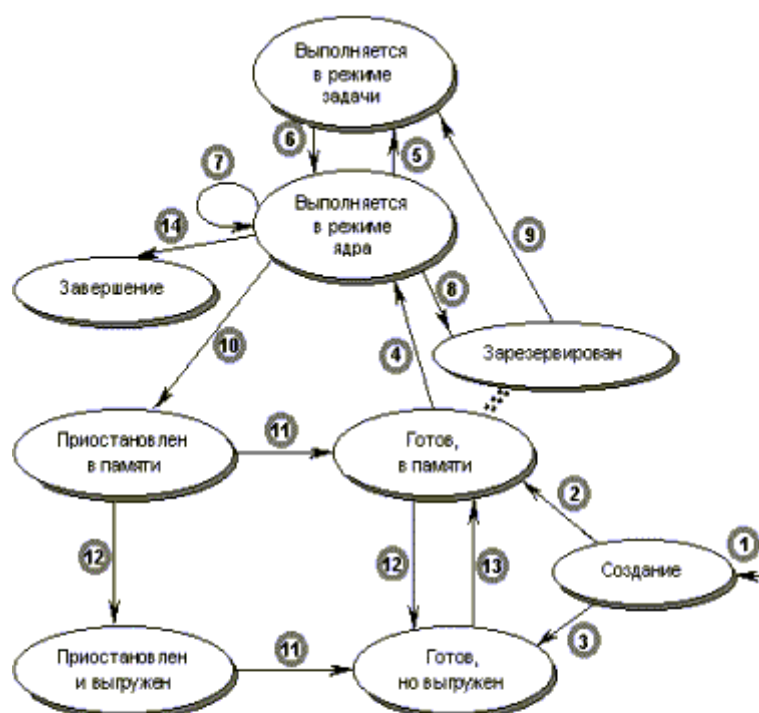
В ОС Tru64 Unix имеется вызов `pthread_atfork()`, который позволяет определить процедуру, выполняемую в процессе, порождаемом системным вызовом `fork()`.

Помимо пользовательских нитей, почти во всех описанных нами ОС бывают еще нити ядра, порождаемые с помощью вызова `kernel_thread()` для внутренних системных нужд.

6.5. Планирование процессов

Механизм планирование нитей библиотекой нитей является надстройкой над механизмом планирования процессов (легковесных процессов), реализуемых ОС.

Состояния процессов, которые различаются в ОС Unix, показаны на рисунке 6.3.



Переходы между состояниями:

- | | |
|--|--------------------------------------|
| 1. системный вызов fork | 7. прерывание, возврат из прерывания |
| 2. создание процесса (памяти достаточно) | 8. резервирование |
| 3. создание процесса (памяти недостаточно) | 9. возврат в режим задачи |
| 4. запуск | 10. приостановка |
| 5. возврат | 11. возобновление |
| 6. вызов системы, прерывание | 12. выгрузка |
| | 13. подкачка |
| | 14. прекращение |

Рисунок 6.3 Состояния процесса в ОС Unix

Новый процесс создается системным вызовом `fork` (см. ниже) – переход 1 на рисунке 6.3. Если оперативной памяти в системе достаточно, то процесс создается в памяти (выделяется память для размещения адресного пространства процесса) – переход 2. Если памяти недостаточно (только в системе с подкачкой сегментов), то процесс создается на внешней памяти – переход 3. Если процесс создан в памяти, он переходит в состояние готовности к выполнению, из которого может быть запущен на выполнение – переход 4. При запуске процесс начинает выполнение в режиме ядра, но затем "возвращается" (такова специфика выполнения системного вызова `fork` в режим задачи) – переход 5. Выполняемый в

режиме задачи процесс может перейти в режим ядра по внешнему прерыванию или выдав системный вызов (программное прерывание) – переход 6. Выполнение обработчика прерывания или системного вызова происходит в режиме ядра, но в контексте процесса. Выполнение в режиме ядра может прерываться, как было описано выше, но с возвратом в прерванный уровень – переход 7. Если переход в режим ядра произошел по причине прерывания от таймера, и планировщик процессов ОС принимает решение о том, что необходимо запустить другой процесс, то процесс переходит в зарезервированное состояние – переход 8. Зарезервированное состояние в принципе идентично состоянию готовности в памяти, различие между ними состоит в том, что выполнение зарезервированного процесса (когда до него дойдет очередь) возобновляется сразу в режиме задачи – переход 9. Если же переход в режим ядра произошел в связи с выполнением, а это выполнение связано с необходимостью ожидания или планировщик процессов ОС принимает решение о том, что необходимо запустить другой процесс, то процесс переходит в состояние приостановки – переход 10. Когда процесс получит те ресурсы, которых он ожидает, он переводится в состояние готовности – переход 11, из которого он может быть вновь запущен в режиме ядра – переход 4, а из него вернуться в выполнение в состоянии задачи – переход 5. При нехватке памяти приостановленный или даже готовый процесс может быть выгружен на внешнюю память – переходы 12. Процесс, ожидающий ресурса на внешней памяти, там же на внешней памяти может быть переведен в состояние готовности – переход 11. Готовый процесс на внешней памяти (перешедший в состояние готовности из приостановки или созданный на внешней памяти) загружается (подкачивается) в память – переход 13. Когда выполнение процесса завершается (выполнением системного вызова `exit`), процесс переходит в режим ядра, а из него – в состояние завершения – переход 14. В этом состоянии большинство

ресурсов процесса освобождается, но строка в таблице процессов остается, в ней сохраняется некоторая информация о процессе, которая может быть "востребована" и после завершения процесса, например, код завершения процесса.

С самого начала "проект Unix" задумывался как система с разделением времени. Исходная дисциплина планирования процессов, называемая иногда "алгоритмом полураспада" описана нами в разделе 2.3 части I. Там мы, однако, рассматривали только динамическое перевычисление приоритетов и не упомянули о том, что в системе имеются два класса процессов: процессы ядра и пользовательские процессы. Планирование по "алгоритму полураспада" выполняется только для пользовательских процессов. Системные же процессы имеют более высокие приоритеты, и их приоритеты заложены в коде ядра. Так, наивысший приоритет имеет процесс подкачки страниц, следующий – процесс ожидания дискового ввода-вывода и т.д. Даже когда пользовательский процесс выполняется в режиме ядра, он не становится процессом ядра, и никакие динамические добавки к приоритету не могут вывести приоритет пользовательского процесса за тот порог, который разделяет процессы ядра и процессы пользователей. Приоритеты процессов ядра абсолютные: появление процесса ядра, готового к выполнению, прерывает выполнение пользовательского процесса или менее приоритетного процесса ядра.

Дисциплина "алгоритма полураспада" обеспечивает прекрасные показатели справедливости обслуживания, что делает ее удобной для интерактивной работы, но непригодной для выполнения процессов реального времени. Стандарты POSIX и Single Unix Specifications требуют наряду с обеспечением интерактивной работы и обеспечения реального времени. Поэтому в современные клоны ОС Unix дисциплина

планирования несколько модифицирована. В системе теперь различаются три класса процессов (в порядке возрастания приоритетов):

- процессы разделения времени;
- процессы ядра;
- процессы с фиксированным приоритетом.

Процессы разделения времени – аналогичны пользовательским процессам в "старой" ОС Unix. Но цели их планирования несколько изменились. Теперь наряду с обеспечением приемлемого времени реакции для интерактивных процессов дисциплина должна обеспечивать также и хорошую производительность для "счетных" процессов. Для этого алгоритм планирования процессов разделения времени модифицирован изменением размера кванта времени. Для процессов с низким приоритетом размер кванта увеличивается. В соответствии с правилами динамического перевычисления приоритеты тех процессов, которые переходят в состояние приостанова, не исчерпав полученного кванта, повышаются, а тех, которые используют выделенный им квант до конца – понижаются. Это приводит к селекции процессов: интерактивные процессы получают высокие приоритеты, а счетные – низкие. В результате счетный процесс получает квант времени ЦП реже, чем интерактивный, но размер этого кванта для счетного процесса больше.

Процессы ядра – те же, что и в "старой" ОС Unix, и планируются они так же.

Новый класс процессов – процессы с фиксированным приоритетом – во многом похож на процессы ядра. Приоритет для процессов такого класса устанавливается пользователем и не изменяется системой. Сам процесс с фиксированным приоритетом может, однако, изменять свой приоритет во время выполнения. Приоритеты процессов этого класса абсолютные, то есть появление готового к выполнению процесса прерывает выполнение и процесса разделения времени, и процесса ядра, и

процессы с фиксированным, но более низким приоритетом. Для процессов с фиксированным приоритетом пользователь может получить гарантированный порядок их выполнения.

Процесс при запуске наследует класс процесса, породившего его, по умолчанию – это класс разделения времени. Привилегированный пользователь может использовать в своем процессе системные вызовы установки фиксированного приоритета.

Основные системные вызовы, обеспечивающие порождение процессов в Unix, описаны нами в разделе 4.3 части I, это:

- `fork` – порождение процесса-потомка,
- `exec` – смена программы процесса,
- `exit` – завершение процесса,
- `wait` – ожидание завершения процесса-потомка.
- Аналогичные системные вызовы в отношении нитей:
- `th_create` – порождение нити,
- `th_exit` – смена программы процесса,
- `th_join` – ожидание завершения нити.

6.6. Управление памятью

Традиционно API Unix представляет сегментную модель памяти. Каждому процессу при создании выделяется три сегмента: сегмент кодов, сегмент данных и сегмент стека. Процесс может изменять размер выделенного ему сегмента, а также получать сегменты разделяемой памяти и управлять ими.

Дескриптор каждого процесса содержит ссылку на таблицу сегментов, выделенную каждому процессу. Таблица (или связный список) сегментов содержит описатели выделенных процессу сегментов

виртуальной памяти. Дескриптор каждого сегмента содержит адрес (виртуальный) начала сегмента, размер сегмента, возможности доступа (чтение, запись, выполнение) и признак частного или совместно используемого сегмента. Если ОС Unix выполняется на аппаратной платформе, которая поддерживает только сегментную модель динамической трансляции адресов, то дескриптор сегмента содержит также указатель на аппаратный дескриптор сегмента и адрес сегмента в реальной памяти. Если же аппаратная платформа поддерживает сегментно-страничную модель трансляции адресов, то дескриптор сегмента содержит указатель на таблицу страниц данного сегмента. Элемент таблицы страниц описывает виртуальную страницу в адресном пространстве процесса. В нем содержится указатель на соответствующий странице аппаратный дескриптор, указатель на соответствующий странице элемент массива дескрипторов страничных кадров и указатель на соответствующий странице элемент массива дескрипторов дисковых блоков. Из последних двух указателей используются либо один, либо другой. Если виртуальная страница размещена в реальной памяти, используется ссылка на массив дескрипторов страничных кадров (см. ниже). Если же страница вытеснена, то ссылка на массива дескрипторов дисковых блоков используется при поиске страницы на внешней памяти.

Описатели страниц в реальной памяти находятся в массиве дескрипторов страничных кадров. Каждый элемент массива соответствует одному страничному кадру в реальной памяти и содержит копии аппаратных признаков использования страницы, статус страницы, число ссылок на страницу из таблиц виртуальных страниц и несколько указателей, связывающих страничные кадры в списки. Весь массив "прошит" этими указателями в несколько списков:

Список страниц, имеющих постоянный образ на внешней памяти — это страницы, содержащие программные коды или данные, используемые

только для чтения. Образы таких страниц загружаются из программного файла.

Список страниц, содержащих данные, доступные для изменения, их образ на внешней памяти сохраняется в файле или в разделе свопинга и может не совпадать с образом страницы в оперативной памяти.

Список абсолютно свободных страниц, то есть страниц, не включенных в адресное пространство никакого процесса.

Список "условно свободных" страниц, то есть страниц, которые являются кандидатами на вытеснения, элементы этого списка входят также в один из вышеназванных списков.

Для совместно используемых сегментов в таблице сегментов каждого процесса имеется своя запись. В чисто сегментной реализации в этом случае виртуальные сегменты разных процессов связываются с одним и тем же реальным сегментом. В сегментно-страничной реализации виртуальные сегменты разных процессов имеют общую таблицу страниц.

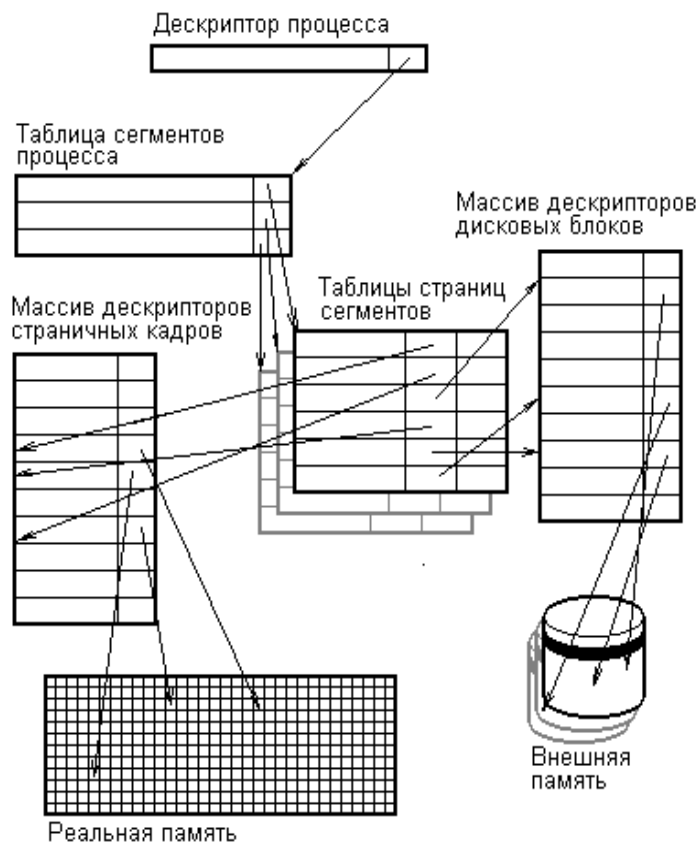


Рисунок 6.4 Логическая структура управления адресным пространством процесса

Представленная на рисунке 6.4 структура управления адресным пространством процесса является логической концепцией, реализация ее на разных аппаратных платформах может существенно различаться. Так, в Open Unix и в других клонах на платформе Intel-Pentium сегментная структура адресного пространства является лишь оболочкой, обеспечиваемой программным окружением, а не ОС. На самом же деле адресное пространство процесса здесь представляет собой просто массив страниц. Сегменты являются только группами страниц, а представляются для процесса в виде традиционных для Unix сегментов только модулями библиотеки системных вызовов.

В реализациях Unix на страничной (или сегментно-страничной) аппаратной платформе свопинг ведется на уровне страниц. Используется концепция "возраста" страницы, представляющая собой один из вариантов

дисциплины LRU. В дескрипторе каждой виртуальной страницы имеется число – ее возраст. ОС ведет список свободных ("условно свободных") страниц в массиве дескрипторов страничных фреймов и в конфигурационных параметрах устанавливается два числа: верхний и нижний пределы размера этого списка. Поддержанием списка свободных страниц в требуемом состоянии занимается системный процесс – сборщик страниц. При возникновении страничных отказов сборщик страниц проверяет и очищает биты использования всех страниц в реальной памяти. Если в промежутке между двумя такими проверками к странице не было обращений (бит использования страницы остался нулевым), то возраст этой страницы увеличивается. Если размер списка свободных страниц достигает нижнего предела, сборщик страниц пополняет его теми страницами, возраст которых наибольший. При этом он доводит размер списка до верхнего предела. Страничный кадр, попавший в список "условно свободных" не сразу распределяется для размещения в нем другой виртуальной страницы. Если до нового распределения к нему будет обращение, то он будет исключен из списка свободных кадров. В последних реализациях Unix алгоритм сборки страниц упрощается: в качестве возраста используется единственный бит – бит использование, а проверка и сброс его производится периодически.

Ранние версии Unix поддерживали две модели свопинга – подкачку страниц (описанную выше) и подкачку процессов. Вторая модель, исторически появившаяся первой, была ориентирована на сегментную модель динамической трансляции адресов в аппаратуре, но вытеснялся при этом весь процесс, всеми своими сегментами. Состояния процесса "приостановлен, вытеснен" и "готов, вытеснен" на рисунке 6.3. связаны именно с моделью подкачки процессов. При повсеместном внедрении аппаратных платформ с сегментно-страничной аппаратной трансляцией адресов произошел отказ от подкачки процессов и свопинг ограничился

только страничным уровнем. Однако в дальнейшем подкачка процессов была совмещена с подкачкой страниц. Так, в Open Unix при нормальной загрузке свопинг осуществляется только на страничном уровне. Однако при существенной нехватке памяти процесс может быть вытеснен целиком, всеми своими страницами. Таким образом, состояния "приостановлен, вытеснен" и "готов, вытеснен" продолжают оставаться актуальными.

Процессы с фиксированными приоритетами никогда не вытесняются полностью, хотя их отдельные страницы могут вытесняться. Но у такого процесса имеется возможность защитить избранные области своего адресного пространства даже от свопинга на уровне страниц.

Современная ОС Unix обеспечивает также файлы, отображаемые в память. Системный вызов `mmap` возвращает адрес в виртуальном адресном пространстве процесса данных открытого файла. Семантика вызова `mmap` такова:

```
paddr=mmap(addr,len,prot,flags,fd,off);
```

где `addr` – "заказываемый" виртуальный адрес сегмента в адресном пространстве процесса (возвращаемое значение может и не совпадать с ним), `len` – размер сегмента, `prot` – возможности доступа, `flags` – управляющие флаги, `fd` – манипулятор файла или другого объекта, `off` – смещение в файле.

Задание смещения и размера отображаемой области дает возможность, если размер файла превышает доступный размер виртуальной памяти, имеется возможность отображать файл частями или создавать "плавающие окна" отображения. Механизм отображения в память применим не только к файлам на внешней памяти, но и к другим объектам, внешние имена которых находятся в пространстве имен файловой системы: таблицам, структурам, общим областям памяти и т.п.

Однако отображаемые объекты обязательно должны быть объектами памяти – оперативной или внешней. Не могут быть отображаемыми файлами, например, терминалы или сетевые соединения.

Механизм отображения в память в Open Unix заменяет старые средства управления памятью. Хотя традиционные системные вызовы управления памятью здесь тоже сохраняются, но они обеспечиваются только библиотекой системных вызовов, которая преобразует их в отображение в память. Механизм отображения в память позволяет процессам также получать дополнительные частные сегменты в адресном пространстве процесса. Отображение в память "файла" с именем `"/dev/zero"` дает процессу частный сегмент запрошенного размера.

Выполнение системного вызова `fork` в традиционной ОС Unix подразумевает откладывание выделения для процесса-потомка отдельных частных сегментов. Вместо этого применяется политика "копирования при записи": два процесса – предок и потомок – разделяют один сегмент, но этот сегмент помечается как защищенный от записи. Пока ни один из процессов не изменяет содержимого сегмента, процессы используют один и тот же сегмент. Когда один из процессов пытается писать в сегмент, происходит исключение, обрабатывая которое ОС создает копию сегмента для процесса-потомка, и с этого момента каждый процесс использует свой собственный сегмент. В Open Unix политика "копирования при записи" реализуется на уровне страниц.

Динамическая компоновка в современных Unix в принципе идентична таковой во многих других современных ОС (OS/2, Windows 9x, Windows NT). Модули динамической компоновки называются здесь разделяемыми объектами (`shared object`). Они представляют собой программные модули, содержащие несколько входных точек. Библиотека разделяемых объектов, однако, содержит не один такой модуль, а целый набор их. При загрузке или при выполнении кодовый сегмент

разделяемого объекта включается в адресное пространство процесса. Код разделяемого объекта совместно используется всеми процессами, но отображается в общем случае в разные участки виртуальной памяти разных процессов. Разделяемые объекты, таким образом, компилируются и компонуются так, чтобы они могли быть перемещаемыми в виртуальном адресном пространстве. Для переменных разделяемого объекта создается отдельная область для каждого процесса, использующего объект.

6.7. Средства взаимодействия процессов

Все средства взаимодействия процессов, описанные нами в главе 9 части I, обеспечиваются в ОС Unix.

Программные каналы являются средством обеспечения взаимодействия между процессами. В старых реализациях Unix обеспечивались классические межпрограммные каналы, неименованные и именованные, описанные в разделе 9.6 части I. В новых версиях базовый интерфейс каналов остался тем же, но их внутренние механизмы изменились. Теперь программные каналы называются базирующимися на потоках и при создании канала создаются два разнонаправленных потока, по которым данные движутся по правилам FIFO.

Сигналы в Unix являются не столько средством взаимодействия между процессами, сколько средством взаимодействия между ядром и процессами. Сигналы посылаются ядром процессу, чтобы сообщить процессу о некотором событии, чаще всего представляющем собой возникновение некоторой нештатной ситуации. В Open Unix имеется 31 тип сигналов, которые можно разделить на группы:

- сигналы, связанные с оборудованием, такие как: сигналы об ошибках на шинах передачи данных, ошибках при выполнении арифметических операций, ошибках доступа к памяти и т.п.;

- сигналы, связанные с программными событиями: сигнал завершения процесса, сигналы от интервального таймера, сигналы, которыми процессы обмениваются между собой и т.п.;
- сигналы, связанные с вводом-выводом, сигнализирующие о событиях в потоках ввода-вывода;
- сигналы управления заданиями: приостановки и пуска процессов в группе, сигналы о событиях в процессе-потомке и т.п.;
- сигналы управления ресурсами, сообщающие о превышении процессом лимитов использования ресурсов;
- сигналы о событиях на легковесных процессах, обрабатываемые модулями библиотеки нитей.

Разделяемые области памяти являются весьма широко применяемым средством обмена данными между процессами. Один процесс создает разделяемую область памяти при помощи системного вызова `shmget`, другие – присоединяют область памяти к своему виртуальному адресному пространству при помощи системного вызова `shmat`.

Набор семафоров в Unix представляет собой множество традиционных общих семафоров, число которых задается при создании набора семафоров в системном вызове `semget`. Процесс, создавший набор семафоров, может выполнять над ним любые операции, и он определяет права управления любым семафором из набора (при помощи системного вызова `semctl`) для других процессов, использующих семафор. Каждый отдельный семафор состоит из следующих элементов:

- значение семафора,
- идентификатор последнего из процессов, работавших с семафором,
- количество процессов, ожидающих увеличения значения семафора,

- количество процессов, ожидающих момента, когда значение семафора станет равным 0.

Выполнение собственно семафорных операций над любым семафором из набора обеспечивается системным вызовом `semop`.

Возможные операции над семафором:

- увеличение значения семафора на заданное число с выводом из состояния приостановки всех процессов, ожидающих, когда значение семафора станет отличным от нуля;
- уменьшение значения семафора на заданное число, если при этом значение семафора становится равным нулю, выводятся из состояния приостановки все процессы, ожидающие этого события;
- проверка значения семафора, если оно равно нулю, увеличивается число приостановленных процессов, ожидающих, когда значение семафора станет нулевым.

Очереди сообщений в Unix полностью совпадают с очередями, описанными нами в разделе 9.7 части I. Тела сообщений сохраняются в адресном пространстве ядра. Заголовки сообщений в очереди составляют связный список, каждый элемент которого содержит:

- указатель на следующее сообщение в списке;
- тип сообщения;
- размер тела сообщения;
- адрес тела сообщения.
- Каждая очередь описывается структурой, содержащей:
- указатели на первое и последнее сообщения в очереди;
- текущее число байт и текущее число сообщений в очереди;
- максимальное число байт в очереди;

- идентификаторы процессов, пославшего и получившего последнее сообщение в очередь/из очереди;
- времена последней посылки сообщения и последнего получения сообщения.

Введение нитей потребовало также и введения средств синхронизации выполнения нитей внутри одного процесса. Системная библиотека нитей обеспечивает ряд специальных средств синхронизации. Эти средства используются только внутри одного процесса. Средства синхронизации нитей включают в себя:

Замок взаимного исключения (mutual exclusion lock). Операции установки замка и снятия замка идентичны скобкам критической секции.

Зацикленный замок (spin lock) – аналогичен простому замку, но вместо ожидания в приостановке обеспечивает занятое ожидание. Такие замки защищают нить от вытеснения с легковесного процесса при ожидании на замке.

Рекурсивный замок (recursive lock) – аналогичен простому замку, но исключает самоблокирование нити при рекурсивном входе в критическую секцию.

Замок чтения-записи (reader-writer lock) – обеспечивает нахождение в критической секции в каждый момент времени не более одной нити-"читателя" и любого числа нитей-"писателей".

Семафор – традиционный общий семафор с несколькими ограниченными возможностями управления по сравнению с семафорами, применяемыми при взаимодействии процессов.

Барьер – средство, обратное замкам. Барьер создает "критическую секцию", в которую может войти только заданное количество нитей одновременно.

Переменная условия (condition variable) – аналог событий. Обеспечивает перевод нити в ожидания до тех пор, пока эта переменная не будет установлена в другой нити.

При освобождении нитей, ожидающих на средствах синхронизации, применяются следующие правила:

- связанные нити пользуются абсолютным преимуществом перед мультиплексируемыми;
- для освобождения выбирается нить с наивысшим приоритетом;
- при равенстве приоритетов нити освобождаются в порядке FIFO.

Механизмы синхронизации нитей не применяются автоматически, ответственность за применение этих средств и за предупреждение тупиков несет программист. Для применения каждого из описанных ниже средств синхронизации программист должен создать средство (выделить память для размещения соответствующей структуры), инициализировать его, использовать его, освободить его.

6.8. Файловые системы

Базовой единицей, в которую Unix записывает информацию, является файл. Файл – это именованное собрание данных, которое является единицей хранения данных. Файлы записываются в область памяти на одном или нескольких дисках, которая называется файловой системой. Файловая система разделяется на меньшие области памяти, которые называются каталогами. Каталоги могут включать в себя файлы и другие каталоги и образуют иерархическую структуру с общим корнем. В отличие от ряда других ОС (MS DOS, OS/2, все Windows) в единое дерево каталогов в Unix включаются все тома дисковых носителей.

Корневым каталогом является каталог с именем "/". В этом каталоге находятся файлы, используемые при загрузке системы и другие каталоги.

Некоторые каталоги в корневом каталоге создаются при инсталляции системы. Среди этих каталогов:

- `/bin` и `/usr/bin` – каталоги, содержащие большинство команд ОС Unix. Обычно стандартные команды находятся в каталоге `/bin`, а команды, специфические для определенной группы пользователей, – в `/usr/bin`.
- `/dev` – каталог, содержащий специальные файлы устройств, используемые для доступа к внешним устройствам.
- `/etc` – каталог файлов системной конфигурации и команд системного администратора
- `/unix` – каталог программ ядра ОС. Эти программы загружаются в память при запуске системы.
- `/usr/lib` – каталог файлов прикладных библиотек.
- `/usr/spool` – каталог временных файлов и очередей.
- `/var/opt/` – каталог символьных связей.

Каждый том представляет собой отдельную файловую систему со своим деревом каталогов. При работе с несколькими томами файловая система каждого тома подключается (монтируется) к общей файловой системе в виде ветви общего дерева каталогов. Монтирование не обязательно производится к корню общей файловой системы.

ОС Unix обеспечивает также те средства, которые мы в главе 7 части I описали как алиасы и косвенные файлы. Алиасы – в Unix они называются жесткими связями (`hard link`) или просто связями – представляют собой две или более ссылок на один и тот же физический файл из разных каталогов (или из одного каталога, но под разными именами). Они легко обеспечиваются в ОС Unix за счет того, что в большинстве файловых систем Unix (см. ниже) дескриптор физического файла и элемент каталога хранятся раздельно. Косвенные файлы в Unix называются символическими

связями (symbolic link). Физически символическая связь представляет собой специальный файл.

Наряду с интерпретацией файловой системы как общего (многоотомного) пространства памяти с логической структурой дерева каталогов, в Unix под файловой системой понимают также и отдельный том с его деревом каталогов и со специфической физической структурой хранения данных и управления дисковым пространством. Говорят о различных типах файловых систем, имея в виду тома с различной физической структурой хранения. ОС Unix позволяет монтировать различные типы файловых систем в общую логическую структуру. Некоторые из типов файловых систем мы рассматриваем ниже.

Файловая система s5

Файловая система s5 была исторически первой файловой системой в Unix. Структура тома в файловой системе s5 показана на рисунке 6.5.

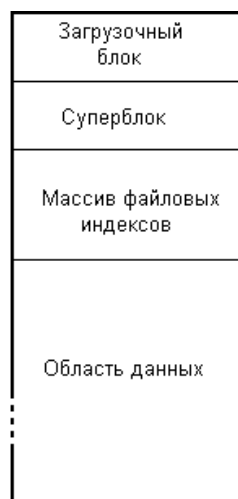


Рисунок 6.5 Структура тома в файловой системе s5

Загрузочный блок содержит программу начальной загрузки системы, в несистемных дисках этот блок присутствует, но не используется.

Суперблок является самой важной системной структурой на диске. Он содержит основную информацию о томе, в том числе имя тома и размер логического блока на томе. В суперблоке содержится также общее число файловых индексов на томе, массив из 100 номеров свободных индексов и индекс, с которого начинается очередной поиск в массиве индексов. При необходимости создания индекса для нового файла номер свободного индекса выбирается из этого массива. Если же массив свободных индексов в суперблоке будет исчерпан, просматривается массив индексов, и номера первых 100 найденных свободных индексов заносятся в массив номеров в суперблоке.

Также в суперблоке имеется массив из 50 номеров свободных логических блоков в области данных. Последний элемент этого массива адресует логический блок, в котором находится продолжение списка свободных блоков, а последний элемент в нем – блок с продолжением списка и т.д.

Файловые индексы в Unix представляют собой то, что мы в главе 7 части I называли дескриптором файла. Оригинальное название такого индекса – inode – i-узел. В файловой системе s5 дескрипторы отделены от элементов каталогов. Файловый индекс содержит информацию о физическом файле, а именно:

- тип файла (обычный файл, каталог, специальный файл);
- количество связей (алиасов) файла;
- идентификатор пользователя-владельца и группы владельца;
- размер файла;
- план размещения файла.

Специальными файлами в s5 (и в других файловых системах Unix) являются блочные и символьные устройства, именованные каналы, символьные связи (косвенные файлы).

Файловый индекс не содержит имени файла – оно содержится в элементе каталога. Максимальное число файловых индексов, возможное на томе s5 – 65500

План размещения файла в s5 описан нами в разделе 7.6 части I, он представляет собой несбалансированное дерево с прямой адресацией к начальной части файла и одно-, двух- и трехуровневой адресацией к следующим частям. Отметим, что в современных версиях файловой системы s5 единицей распределения дискового пространства является не физический блок размером 512 байт, а логический блок, размер которого может выбираться из ряда 512, 1024, 2048 байт.

В области данных располагаются файлы и каталоги.

Каталоги s5 состоят из 16-байтных элементов, в которых первые 2 байта – номер индекса в массиве файловых индексов, а еще 14 байт – имя файла.

Файловая система ufs

Файловая система ufs пришла из BSD Unix и имеет ряд существенных отличий от s5. Первое отличие состоит в том, что пространство диска в ufs разбивается на участки одинакового размера, называемые группами цилиндров, и управляющие структуры распределены по группам цилиндров, как показано на рисунке 6.6.

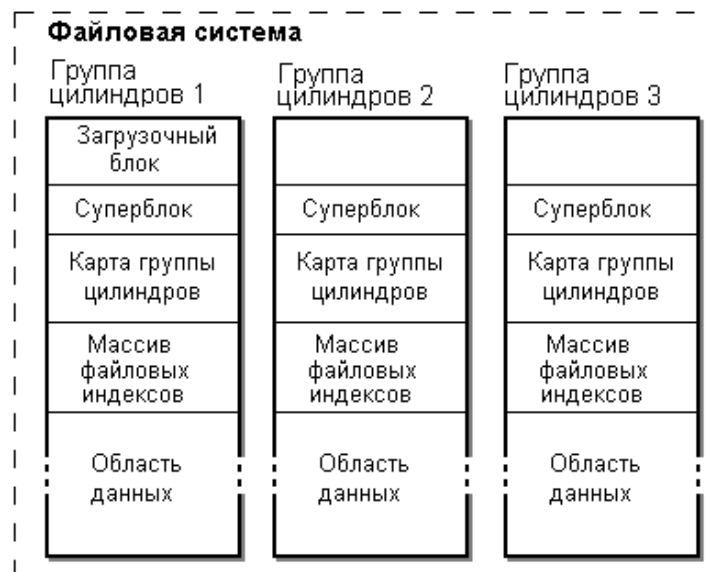


Рисунок 6.6 Структура тома в файловой системе ufs

Загрузочный (его размер – 8 Кбайт) блок присутствует только в первой группе цилиндров. В каждой группе цилиндров имеется копия суперблока. При нормальной работе используется только суперблок из первой группы цилиндров, остальные копии служат для восстановления суперблока в случае порчи оригинала.

В суперблоке ufs содержится:

- размер и состояние файловой системы,
- метка (имя) тома,
- время последнего изменения,
- размер группы цилиндров.

Карта группы цилиндров содержит информацию о состоянии (занят/свободен) логических блоков в области данных.

Назначение массива файловых индексов в ufs – такое же, как и в s5, но массив каждой группы цилиндров относится только к файлам данной группы. В файловом индексе ufs содержится:

- тип файла (обычный файл, каталог и т.д.),
- режим доступа (чтение, запись, выполнение),

- количество связей файла;
- идентификатор пользователя-владельца и группы владельца;
- размер файла;
- время создания файла, последнего доступа к файлу, последней модификации файла;
- план размещения файла.

План размещения файла в принципе такой же, как и в s5, но здесь он размещается в двух массивах. Первый массив содержит 12 прямых ссылок на логические блоки, содержащие данные. Второй массив содержит три элемента, являющиеся вершинами одно-, двух- и трехуровневых деревьев ссылок соответственно.

Размер логического блока выбирается из ряда: 2, 4, 8 Кбайт. Поскольку большой размер блока может приводить к появлению значительных внутренних дыр, допускается разбиение блока на фрагменты. Размер фрагмента может быть 512, 1024, 2048 или 4096 байт. Фрагментированные блоки используются для хранения "остатков" нескольких файлов в одном блоке.

Файловая система ufs позволяет также устанавливать для пользователей квоты на использование двух основных ресурсов файловой системы: файловых индексов и блоков данных. Первый лимит ограничивает количество файлов, создаваемых пользователем, второй – общий объем дисковой памяти для пользователя. Квоты могут быть установлены "жесткие" или "мягкие". Лимиты "жесткой" квоты не могут быть превышены ни при каких обстоятельствах. Лимиты "мягкой" квоты могут быть превышены, но не более, чем на заданный интервал времени.

Файловая система sfs

Файловая система sfs является вариантом системы ufs. Ее отличие состоит в том, что для каждого файла создается два файловых индекса:

- первый индекс (с четным номером) идентичен файловому индексу ufs,
- второй индекс (с нечетным номером) содержит информацию, связанную с безопасностью).

Элемент каталога sfs содержит ссылку только на четный индекс, поэтому программы, работающие с файловыми индексами, "видят" только четный индекс.

Файловая система Veritas

Файловую систему vxfs называют также Veritas – по названию фирмы, выпускающей этот продукт. Достоинствами vxfs являются:

- быстрое восстановление файловой системы после сбоев;
- распределение памяти экстендами;
- расширенные возможности поддержания целостности данных.

Возможность быстрого восстановления системы достигается за счет протоколирования операций, изменяющих метаданные файловой системы в так называемом журнале намерений (intent log). Выполнение каждой прикладной операции в файловой системе может потребовать нескольких изменений в метаданных файловой системы. Так, например, создание нового файла при условии, что в каталоге уже нет свободного места, потребует:

- изменения в карте свободной памяти для выделения нового блока для каталога;
- изменения в блоке каталога;
- изменения в файловом индексе каталога;
- изменения в файловом индексе для нового файла;

- изменения в карте свободных файловых индексов.

Все эти операции должны выполняться как одна транзакция – или выполняться все до конца, или не выполняться вообще. Список намерений представляет собой циклически используемый список. По умолчанию для этого списка выделяется 512 блоков. Любое изменение в структуре файловой системы разбивается на список подопераций и в список намерений заносится запись о транзакции с указанием в ней всех составляющих подопераций. vxfs поддерживает список для всех незавершенных транзакций файловой системы. Запись заносится в список намерений до выполнения входящих в транзакцию подопераций. После того, как запись списка намерений сохранена на диске, подоперации транзакции могут выполняться с любой задержкой. При восстановлении после сбоя системы утилита `fsck` просматривает список намерений и либо завершает, либо откатывает те операции, которые выполнялись во время сбоя.

Логический блок (единица распределения памяти) в vxfs может иметь размер 1, 2, 4, 8 Кбайт. Память для файлов распределяется экстендами – участками, состоящими из одного или более логических блоков. Элемент плана размещения файла содержит две составляющих: номер первого блока в экстенде и размер экстента. План размещения файла в vxfs содержит массив, в котором первые 10 элементов являются прямыми описателями первых экстентов файла, 11-й элемент описывает экстент (по умолчанию размер этого экстента – 8 Кбайт), который используется для косвенной адресации следующих экстентов, 12-й элемент описывает экстент, который используется для двухуровневой косвенной адресации следующих экстентов.

Поддержание целостности прикладных данных является возможностью по выбору. Она обеспечивается тем, что для файловой системы можно установить режим принудительной записи содержимого

кеша на диск при закрытии файла. Разумеется, это снижает производительность файловой системы, и эту возможность рекомендуют использовать только в настольных инсталляциях, где велика вероятность некорректного завершения работы системы.

В настоящее время vxfs поддерживается в версиях 1, 2 и 4. Между версией 1 и последующими есть принципиальная разница, поэтому сначала изложим основные свойства версии 1.

Структура файловой системы vxfs v.1 показана на рисунке 6.7.



Рисунок 6.7 Структура файловой системы vxfs v.1

Общая для всей файловой системы информация включает в себя суперблок и описанный выше список намерений. Остальное пространство диска в vxfs v.1 разбивается на так называемые единицы распределения (allocation unit), смысл которых – тот же, что и групп цилиндров в ufs. В каждой единице распределения содержится:

- Копия суперблока (используется только при порче оригинала). В суперблоке содержится тип и имя файловой системы, дата создания и модификации, информация об общем размере файловой системы и об объеме свободных ресурсов и т.п.

- Общая информация о единице распределения – общее число файловых индексов с расширенными операциями, число свободных файловых индексов, число свободных экстенгов.
- Битовая карта свободных файловых индексов.
- Битовая карта расширенных операций. Это битовая карта файловых индексов, в ней отмечаются те индексы, операции над которыми остаются незавершенными слишком долго, чтобы хранить их в списке намерений.
- Карта свободных экстенгов представляет собой несколько битовых карт разного "масштаба", соответствующих свободным экстенгам размером в 2, 4, 8 Кбайт и т.д.
- Массив файловых индексов. Содержимое файлового индекса качественно то же, что и в ufs. План размещения файла, содержащийся в файловом индексе описан нами выше.

Структура файловой системы vxfs v.1 показана на рисунке 6.8.

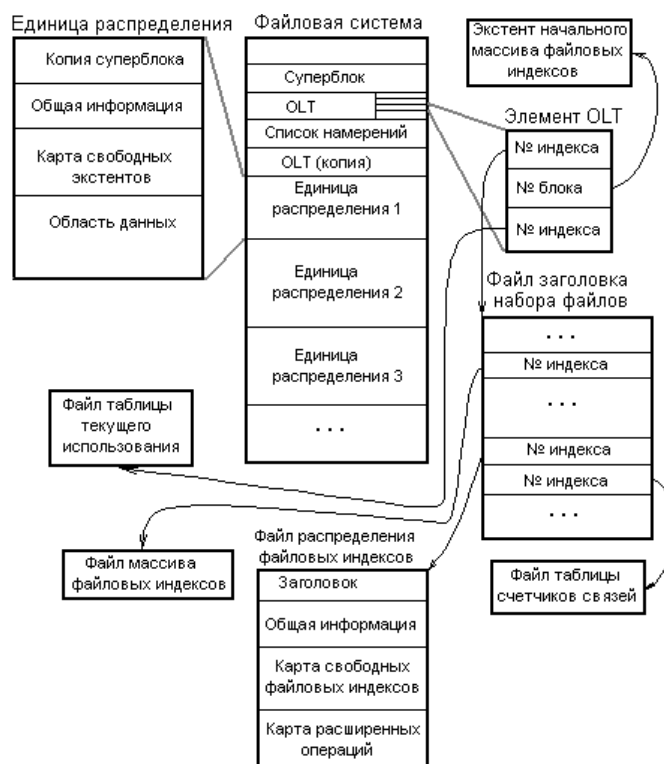


Рисунок 6.8 Структура файловой системы vxfs v.2

vxfs v.2 отличается прежде всего тем, что часть метаданных файловой системы хранится не в статических областях, а в так называемых структурных файлах, память для которых распределяется динамически, как и для обычных файлов. Информация о свободном пространстве хранится в статических структурах единиц распределения. Информация о файлах – в динамических структурах. vxfs v.2 использует концепцию файлового набора (fileset). Имеется два вида файловых наборов:

- файловые наборы атрибутов, содержащие структурные файлы, в которых хранятся метаданные файловой системы; эти файлы не видны пользователям;
- неименованные файловые наборы, содержащие файлы данных, которые видны и доступны пользователям.

Общая для всей файловой системы статическая информация состоит из:

- суперблока (такой же, как в v.1);
- таблицы размещения объектов (Object Location Table – OLT);
- списка намерений (такого же, как в v.1);
- копии OLT.

В каждой единице распределения имеется:

- копия суперблока;
- общая информация (такая же, как в v.1);
- карта свободных экстенгов (такая же, как в v.1);
- область данных.

Доступ к динамически распределяемым структурам производится через таблицу размещения объектов. Эта таблица состоит из элементов, в

каждом элементе таблицы содержится информация об одном файловом наборе, а именно:

- номер файлового индекса для файла, в котором находится заголовок файлового набора;
- адрес блока из двух 8-Кбайтных экстентов, содержащего начальные массив файловых индексов, в этом массиве содержатся индексы всех структурных файлов, описывающих файловый набор;
- номер файлового индекса для файла, в котором находится таблица текущего использования файлового набора.

В файле заголовка файлового набора записаны:

- номер и имя набора;
- файловый индекс файла массивы файловых индексов;
- число выделенных файловых индексов;
- максимально возможное число файловых индексов;
- размер экстента в массиве файловых индексов;
- файловый индекс файла распределения файловых индексов;
- файловый индекс файла таблицы счетчиков связей.

Массив файловых индексов расположен в структурном файле, который занимает несколько экстентов, размер каждого из которых по умолчанию составляет 8. Для этого файла создается копия. Структура файлового индекса – та же, что и в v.1. Файл распределения файловых индексов содержит:

- заголовок, с сигнатурой типа файла;
- общую информацию: число свободных индексов, число индексов с незаконченными операциями и т.п.;
- битовую карту свободных файловых индексов (такую же, как в v.1);

- битовую карту расширенных операций(такую же, как в v.1).

Таблица текущего использования содержит оперативную информацию об использовании файлового набора. Информация в этой таблице может быть воссоздана по другим метаданным файлового набора.

Таблица счетчиков связей содержит число жестких связей для каждого файла в файловом наборе. Индексация в этой таблице совпадает с индексацией в массиве файловых индексов.

vxfs v.4 отличается от v.2 тем, что поддерживает файлы большого (до 2 Гбайт) размера и квоты на число файловых индексов и число блоков данных.

Другие файловые системы Open Unix

Кроме описанных выше, в Open Unix поддерживаются также файловые системы:

- bfs – специализированная файловая система для загрузки и выполнения отдельных программ;
- dosfx – файловые системы FAT (FAT-12, FAT-16, FAT-32 и VFAT);
- memfs – файловая система для виртуальных дисков в памяти;
- cdfs – файловая система CD-ROM.
- nfs – сетевая файловая система

Файловая система JFS

ОС IBM AIX работает также с файловыми системами JFS (эта файловая система перенесена также в OS/2 v.5) и JFS2.

О JFS мы расскажем вкратце, так как она во многом похожа на файловые системы, рассмотренные нами выше. Дискковая память выделяется логическими блоками, размер которых может выбираться от

512 байт до 4 Кбайт. Предусматриваются также "частично заполненные" блоки меньшего размера, в которых хранятся последние части файлов, если они слишком малы для выделения им полного блока. Файловый дескриптор содержит 8 прямых адресов блоков с данными файла и 9-й – адрес блока, через который осуществляется косвенная адресация к остальным данным файла – одноуровневая или (для очень больших файлов) двухуровневая.

Несколько более подробно рассмотрим файловую систему JFS2. Загрузочный блок JFS2 имеет размер 4 Кбайт и располагается, конечно, в первых секторах диска. Однако суперблок (его размер также 4 Кбайт) располагается со смещением 32 Кбайт от начала диска. В суперблоке содержится:

- размер файловой системы;
- число блоков данных в файловой системе;
- состояние файловой системы;
- размер группы распределения;
- размер логического блока.

В файловой системе имеются две битовые карты, используемые для отслеживания состояния и распределения памяти:

- карта для файловых индексов;
- карта для блоков данных.

Группы распределения являются чисто условным понятием, не поддерживаемым какими-либо конкретными структурами. Они используются файловой системой для того, чтобы поместить в соседние области дисковой памяти те данные, доступ к которым осуществляется, как правило, совместно, и разнести несвязанные данные. Таким образом повышается степень локализации обращений к диску.

Единицей дисковой памяти также является логический блок, размер которого выбирается от 512 байт до 4 Кбайт. Но дисковое пространство для файла выделяется экстендами. Экстент занимает непрерывную область дисковой памяти, состоящую из целого числа логических блоков. Большой экстент может даже находиться в нескольких группах распределения. В плане размещения файла экстент представляется номером первого блока и количеством блоков в экстенде, которое может принимать значения от 1 до $2^{24}-1$.

Каждый файловый индекс имеет уникальный во всей файловой системе номер и размер 512 байт. Файловые индексы выделяются системой динамически. В каждом файловом индексе содержится:

- тип файла;
- размер файла и число выделенных файлу блоков;
- информация о правах доступа;
- число алиасов для файла;
- времена доступа и модификации;
- данные файла или план размещения файла.

Для небольших файлов сами данные файла содержатся во второй части файлового индекса. Для больших файлов вторая часть индекса содержит корень B^+ -дерева, через которое адресуются экстенды файла.

Также в B^+ -дерева структурированы каталоги. Элемент каталога имеет размер 512 байт и содержит имя файла и номер файлового индекса для файла. Имеющиеся в любом каталоге элементы с именами "." и ".." записаны не в элементы каталога, но хранятся в самом файловом индексе каталога.

Драйверы устройств во всех Unix являются частью ядра, а не пользовательскими процессами. Доступ к драйверу на пользовательском уровне возможен через специальные файлы устройств, находящиеся в

каталоге /dev. Ядро преобразует файловые операции, выполняемые над этими файлами в обращения к драйверу.

В системе ввода-вывода различаются два типа устройств – блочные и символьные.

6.9. Интерфейсы Unix

При своем рождении ОС Unix была системой командной строки (в соответствии с тогдашним состоянием аппаратуры отображения), в значительной степени таковой она остается и в настоящее время. Это обусловлено как традициями пользователей Unix, но в еще большей степени – богатыми возможностями командного языка ОС. Командный интерпретатор Unix является процессом, работающим в пользовательском режиме, поэтому он может быть легко заменен. В связи с этим командные интерпретаторы интенсивно развиваются все время существования этой ОС, и в настоящее время существует большое число командных интерпретаторов, которые, по-видимому, можно отнести к трем основным семействам:

- Bourn-shell – один из первых командных интерпретаторов для ОС Unix
- C-shell – командный интерпретатор, синтаксис языка которого похож на язык программирования C;
- Korn-shell – развитие Bourn-shell

Названные командные интерпретаторы являются базовыми, на их основе существует множество версий и вариаций. Все командные интерпретаторы ОС Unix объединяют общие основные свойства, такие как:

- все они работают в режиме интерпретации;

- возможность перенаправления ввода-вывода и конвейеризации команд;
- возможность порождения и параллельного выполнения процессов в одном сеансе;
- полный набор алгоритмических возможностей.

Легкость порождения новых процессов в Unix и наличие в системе большого числа утилит, выполняющих прежде всего обработку текстов, делают язык командного интерпретатора не только языком управления системой, но и языком обработки данных. Обязательные возможности командного интерпретатора и обязательный набор утилит Unix-систем определяются стандартом POSIX. Если командный язык не используется как язык программирования, то коды управляющих процедур (скриптов) не зависят от варианта командного языка.

Со временем, однако, у ОС Unix стали появляться и WIMP-интерфейсы. Основой для создания таких интерфейсов является система X Window, разработанная в Массачусетском технологическом институте. Система X Window строится по схеме клиент/сервер. Основой системы является процесс X-сервер, который выполняется на компьютере конечного пользователя и "знает" ту конкретную аппаратуру отображения, которая установлена на данном рабочем месте. X-сервер принимает и выполняет запросы от X-клиентов – программ, выполняющихся на этом же компьютере или на удаленном. Взаимодействие X-клиента и X-сервера управляется событиями. Событием в системе X является вывод X-клиентом информации на терминал или поступление внешнего события, которое принимается X-сервером, и о котором X-сервер сообщает X-клиенту. Описание двустороннего взаимодействия X-сервера и X-клиента и формата пакетов, которыми они обмениваются, составляет X-протокол. Транспортный уровень для X-протокола прозрачен. Если клиент и сервер находятся на одном компьютере, то для обмена между ними используются

средства локального взаимодействия, если на разных – любые сетевые средства. Система X представляет собой совокупность программ и библиотек, таким образом, при разработке интерфейса программист может и не знать деталей X-протокола.

Консорциумом OSF принят в качестве стандарта для разработки графических интерфейсов пакет Motif, построенный на базе X Window. Пакет включает в себя менеджер окон, набор вспомогательных утилит, а также библиотеку классов. В пакете имеются также широкие возможности для создания новых графических классов.

В большинстве промышленных Unix-систем в настоящее время используется построенный на основе OSF/Motif интерфейс CDE (Common Desktop Environment), в ОС Linux применяется большое количество различных интерфейсных систем, наиболее популярные из которых – KDE и Gnome.

6.10. Unix-системы фирмы Caldera

Новый этап развития Unix и Linux связан с тем, что летом 2001 г. фирма Caldera – одна из ведущих фирм, предлагающих Linux, купила Unix-подразделение фирмы SCO. Из тех заявлений покупателя, которые были сделаны по этому поводу, поначалу можно было заключить (хотя явно об этом не говорилось), что фирма не собирается развивать далее Open Unix, а намерена интегрировать ее технологии в Linux. Однако, последующие действия фирмы опровергли опасения по поводу возможной "кончины" Unix.

Фирма Caldera выступает на рынке программного обеспечения прежде всего как производитель серверных операционных систем для платформы Intel и в настоящее время предлагает три операционные системы [16]:

- Open Linux (текущая версия – 3.1.1) – "коренной" продукт фирмы;
- Open Unix (текущая версия – 8) – прямой наследник AT&T Unix – Novell Unixware – SCO Unixware;
- Open Server (текущая версия – R5) – продолжение линии SCO Open Server.

Эти продукты Caldera охватывают спектр задач от "тонких" клиентов до корпоративных информационных систем.

Open Unix является и будет оставаться стратегическим продуктом фирмы, прежде всего потому, что даже при примерно одном уровне продаж за последние годы, он обеспечивает стабильно растущий доход. Open Unix 8 – основа для работы приложений и серверов в масштабе от рабочих станций до корпоративной информационной системы. Open Unix существует в редакциях: Base, Business, Department, Enterprise, Data Center, каждая из которых отвечает требованиям различных категорий задач (по возрастающей). Во всех редакциях существуют средства (или расширения), позволяющие обеспечить высокую устойчивость, безопасность и почти линейную масштабируемость. Open Unix 8 является ОС, ориентированной прежде всего на решения промышленного масштаба. Наиболее часто Open Unix применяется в качестве платформы для сервера баз данных или сервера электронного бизнеса (сервера приложений, ориентированного на выполнение транзакций). В частности только на Open Unix и Unixware 7 работает Caldera ReliantHA – программное решение, обеспечивающее поддержку кластерных соединений (2 - 4 узла) с эффективным масштабированием производительности, постоянным мониторингом состояния системы и автоматическим перераспределением задач при выходе из строя одного из узлов. Использование кластерных решений с ReliantHA для Open Unix 8 позволяет повысить коэффициент готовности системы до 99.995. Open Unix будет продолжать развиваться,

прежде всего – в направлении обеспечения платформы для крупномасштабных корпоративных решений и будет и в дальнейшем

Интересным свойством Open Unix 8 является интеграция возможностей Unix и Linux. В этой ОС могут выполняться как "родные" Unix-приложения, так и приложения для Open Linux или для других Linux-систем, соответствующие спецификациям Linux Standard Base, причем зачастую – с лучшей производительностью, чем на "родных" Linux-системах. Open Unix 8, таким образом, сочетает в себе надежность и масштабируемость Unix с простотой использования Linux. (О механизме обеспечения такой интеграции – см. ниже.)

Open Linux Server поставляется в трех конфигурациях, готовых к работе с наиболее распространенными серверными программными продуктами:

- Secure Web Server;
- File and Print Server;
- Network Infrastructure.

Open Linux Workstation ориентирована прежде всего на разработку и портирование приложений для Unix и Linux платформ. Включает в себя полный набор средств разработки, среди которых ведущее место занимают средства технологий Java.

Open Linux является программным продуктом, а не дистрибутивом, применение его покупки лицензии. Это также означает, что для этой ОС фирма несет ответственность (в том числе и юридическую) за работу системы, публикует и реализует планы развития, проводит обучение и сертификацию специалистов и разработчиков приложений, обеспечивает средства управления системой. В числе последних фирма Caldera начала выпуск продукта Volution, обеспечивающего интеграцию и централизованное управление распределенными системами на базе Unix и Linux (от разных производителей).

Темпы развития ОС Unix не могут быть слишком высокими, потому что именно эта система была и остается пионером в поиске и внедрении новых технологий, и ей просто "неоткуда ждать подсказок". Темпы развития Linux могут быть сверхвысокими – в первую очередь благодаря заимствованиям из Unix. Хотя эти ОС становятся и будут становиться все более "дружественными" одна к другой, их слияние в ближайшей перспективе не произойдет. Интенсивное развитие Open Linux происходит прежде всего за счет внедрения в нее технологий Open Unix. Такое заимствование позволяет прогнозировать превращение Open Unix в действительно промышленную ОС масштаба предприятия уже в ближайшие год-два.

Open Server R5 является многофункциональной серверной ОС для выполнения широкого спектра задач среднего класса. Хотя эта ОС считается несколько устаревшей (строится на базе более ранней версии ядра, чем большинство других Unix-систем), для нее существует множество приложений, и она пока остается самой популярной Unix системой: 38% инсталляций Unix во всем мире приходится именно на эту ОС. В постсоветских странах ее доля еще выше и составляет 47.5%, причем во многих случаях продолжает эксплуатироваться версия R3. Фирма Caldera рекомендует своим пользователям мигрировать с Open Server на Open Unix, но не будет заставлять их форсировать этот процесс. Open Server будет поддерживаться и развиваться, хотя качественных скачков в развитии этой ОС ожидать уже не приходится.

Важной акцией, обеспечивающей плавную миграцию с Open Server на Open Unix, является введение в Open Unix функции Portable Open Server, обеспечивающей выполнение приложений Open Server в среде Open Unix и Open Linux.

Для обеспечения выполнения приложений Open Linux в среде Open Unix была разработана технология Linux Kernel Personality. Суть ее

состоит в том, что в ядро Open Unix были включены специфические системные сервисы ядра Linux и таким образом обеспечено выполнение системных вызовов Linux. Процессы, составляющие "образ" той или иной ОС для пользователя, работают в пространстве пользователя. Системные вызовы обеих ОС выполняются одним ядром. Таким образом, достигается не эмуляция Linux на Unix, а действительная интеграция обеих ОС на уровне ядра.

Аналогичная технология Open Server Kernel Personality применена и для Open Server, причем процессы, выполняющиеся в разных операционных средах, могут взаимодействовать через обычные средства IPC. Специальный процесс File Update Daemon синхронизирует изменения файлов, сделанные в разных средах (три ОС используют разные файловые системы). Не вполне ясно, как ядро распознает системные вызовы от той или иной среды. Если в случае Linux Kernel Personality это должно быть достаточно просто: Unix использует для системных вызовов gateway, а Linux – `int 80h`, то Open Server также использует gateway, и отличить вызовы этой среды от вызовов Open Unix сложнее.

Таким образом, фирма Caldera является на сегодня ведущим производителем Unix-систем и деятельность фирмы позволяет сделать следующие выводы:

- "классическая" ОС Unix, ведущая свою родословную от системы К.Томпсона, сохраняет лидирующие позиции и продолжает развиваться;
- ОС Linux превращается в промышленную систему, масштабируемую до масштаба предприятия;
- подавляющее большинство производителей аппаратных и программных средств поддерживают платформы Unix и Linux и рассматривают их как наиболее перспективные среды, в которых

будет происходить совместная работа и взаимодействие программного обеспечения от разных производителей.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Назовите известные вам коммерческие и некоммерческие клоны Unix.
2. Что составляет системный и пользовательский контексты процесса?
3. Как в современных версиях Unix поддерживаются процессы реального времени?
4. Опишите две модели обеспечения нитей, применяемые в разных клонах Unix.
5. Какие средства взаимодействия процессов в Unix вам известны?
6. Какова логическая модель памяти в Unix? Как она реализуется в современных версиях этой ОС?
7. Опишите работу процесса-сборщика страниц.
8. Какие средства введены в Unix для обеспечения синхронизации нитей?
9. Почему файловая система s5 не применяется в современных версиях Unix?
10. Каковы общие свойства современных файловых систем Unix?
11. В чем состоит целостность файловой системы, какими современными файловыми системами Unix и какими средствами она обеспечивается?
12. Сопоставьте файловые системы vxfs или JFS и NTFS. Что между ними общего, в чем различия?

Глава 7. Операционные системы "тонких" клиентов

7.1. Карманные персональные компьютеры

Под "тонкими" клиентами понимают вычислительные устройства, обладающие неполной функциональностью. Вообще говоря, спектр таких устройств достаточно велик – от специализированных вычислителей, встраиваемых в бытовую аппаратуру, до "сетевых" компьютеров, обладающих практически всеми аппаратными возможностями ПК, кроме жесткого диска. Мы в этой главе рассматриваем в основном класс тонких клиентов, называемых карманными персональными компьютерами или PDA – Personal Digital Assistant (персональный цифровой помощник). Такие устройства используются в качестве интеллектуальных органайзеров или/и в качестве мобильных устройств доступа к серверам информационных систем. По оценкам некоторых экспертов число мобильных клиентов во всем мире к 2003 г. достигнет 80 млн. Производство карманных ПК является перспективным направлением, и на этом направлении действует большое число фирм. Многие из них разрабатывают собственные ОС для своих PDA, однако многие используют и ОС от других производителей. Практически каждая фирма – производитель карманных ПК придает своей модели некоторые уникальные свойства, дающие ей какие-то конкурентные преимущества. Однако все многообразие карманных ПК, по-видимому, можно свести к двум типам: ПК без клавиатуры, ввод данных в которых осуществляется рукописный или с виртуальной клавиатуры, в обоих случаях – при помощи светового пера, и ПК с клавиатурой. Второй тип приближается к настольным ПК, в частности, в его функциональность включается и

обработка мультимедийной информации. Первый тип беднее (точнее говоря, специфичнее) по функциональности, однако отличается меньшими размерами и энергопотреблением. Задача ОС для ПК первого типа – обеспечить максимальную экономию ресурсов, задача ОС для ПК второго типа – обеспечить максимальную функциональность. Среди универсальных ОС для карманных ПК первого типа лидирует PalmOS, для второго типа – Windows CE.

7.2. Операционная система PalmOS

Операционная система PalmOS [31] предназначена для управления PDA на базе микропроцессора Motorola Dragon Ball VZ, за которыми закрепилось название PalmPilot (хотя правильное название их – просто Palm). Однако архитектура устройства Palm – открытая и многие фирмы выпускают собственные PDA, подобные Palm, но с теми или иными отличиями – Sony, HandEra, Kyocera, Symbol и другие. Все эти PDA работают под управлением PalmOS.

Специфика функционирования приложений в PalmOS, а, следовательно, и самой PalmOS заключается в следующем:

- малый размер экрана (160x160 точек) не позволяет приложению иметь сложный интерфейс; при проектировании интерфейса следует соблюдать баланс между информационной достаточностью и перегруженностью экрана;
- приложение должно иметь простую и быструю навигацию, выбор требуемого действия должен производиться одной-двумя операциями пользователя, а не длинным диалогом, как бывает в настольных ПК;

- ОС и приложения функционируют в условиях очень ограниченного объема ресурсов, прежде всего – памяти;
- одним из важнейших требований является эффективное управление питанием.
- ОС должна быть рассчитана на быстрый рост вычислительных возможностей – как собственных базовых возможностей PDA, так и расширения номенклатуры, возможностей и форматов подключаемых к PDA карт.

Обязательным компонентом платформы Palm является синхронизационная приставка (cradle), которая обеспечивает соединение с настольным ПК и синхронизацию данных, находящихся на ПК и на PDA. Многие приложения для PalmOS имеют аналоги для настольного ПК. Для синхронизации данных, разделяемых ПК и PDA, используется технология HotSync, которая предусматривает создание специальных каналов (conduit) для синхронизации данных. Существуют специальные инструментальные средства для программирования таких каналов, и многие популярные программные продукты (Netscape Communicator, Oracle, IBM DB2, etc.) имеют в своем составе такие каналы.

Архитектура PalmOS показана на рисунке 7.1.

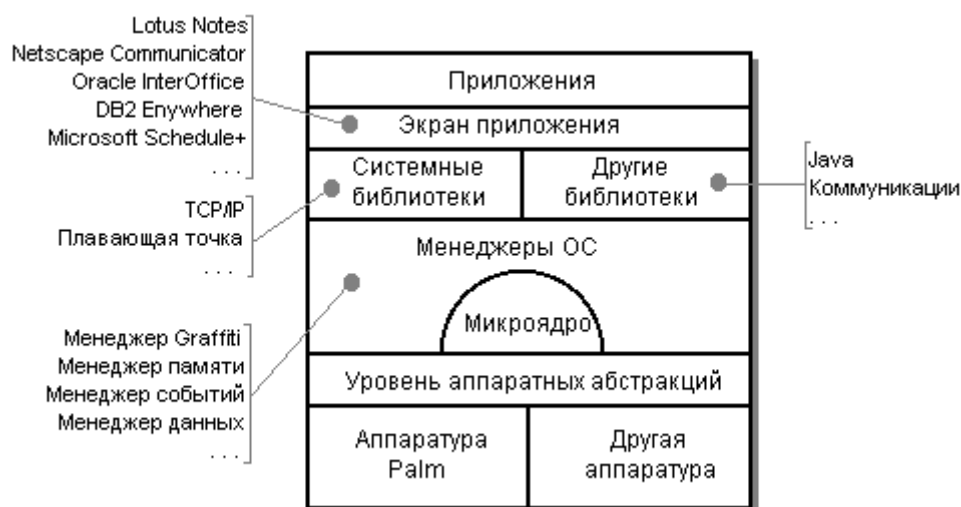


Рисунок 7.1 Архитектура PalmOS

PalmOS базируется на микроядре. Вокруг микроядра построены системные службы – менеджеры PalmOS, представляющие собой наборы модулей, обеспечивающих определенные функции. К таким менеджерам относятся, например:

- Менеджер Graffiti – система рукописного ввода;
- Менеджер Событий;
- Менеджер Памяти;
- Менеджер Данных;
- Менеджер Ресурсов;
- Менеджер Звука;
- и т.д.

Библиотеки (системные или от других производителей) обеспечивают высокоуровневый доступ к системным функциям. Хотя для PalmOS и существуют стандартные библиотеки языка C, в системных библиотеках имеются собственные функциональные аналоги стандартных функций C, которые оптимизированы для PalmOS, поэтому их использование предпочтительнее.

Микроядро и управление задачами

Само микроядро не является собственностью фирмы Palm, используется микроядро AMX RTOS [13], разработанное фирмой Kadal. Микроядро AMX RTOS представляет собой микроядро реального времени, адаптированное для нескольких аппаратных платформ и приспособленное для размещения в ПЗУ. Микроядро обеспечивает вытесняющую многозадачность с абсолютными приоритетами. В каждый момент времени выполняется только задача с наивысшим приоритетом. Переключение задач может происходить

- по инициативе самой задачи – задача переходит в состояние ожидания или запросить выполнение операции, вызывающей выполнение задачи с более высоким приоритетом;
- по внешнему событию – прерыванию, обработка которого может запустить или "разбудить" задачи с более высоким приоритетом;
- по событию таймера, которое также может "разбудить" задачи с более высоким приоритетом.

Микроядро AMX RTOS оптимизировано с целью минимизации времени переключения задач и минимизации нерезидентных участков кода. Как правило, прерывания могут обрабатываться даже во время переключения задач. Служба обработки прерывания микроядра обеспечивает прием прерывания и вызов пользовательской процедуры обработки прерывания. AMX RTOS обеспечивает также вложенную обработку прерываний для тех аппаратных платформ, на которых вложенные прерывания поддерживаются процессором.

Микроядро обеспечивает также богатые средства синхронизации процессов:

- события – события могут образовывать группы до 16 событий, и ожидаться может любая заданная конфигурация совершенности событий в группе;
- семафоры – традиционные общие семафоры, используемые как счетчики ресурсов;
- очереди сообщений – "почтовые ящики" (одна очередь) и "коммуникационные каналы" (4 очереди с разными приоритетами).

Также микроядро обеспечивает ряд сервисных функций, таких как выделение/освобождение памяти, работу с пулом буферов и работу со связными списками.

Микроядро написано на языке С и, следовательно, может быть реализовано для любой аппаратной платформы. Средства конфигурирования позволяют включать в микроядро только те функции, которые необходимы заказчику.

Микроядро само по себе обеспечивает вытесняющую многозадачность. Но по условиям лицензионного соглашения на использование микроядра API микроядра является закрытым, и разработчики не могут создавать программы, напрямую использующие функции микроядра, в том числе и многозадачность. Поэтому приложения PalmOS – однозадачные. В PalmOS в каждый момент времени может выполняться только одно приложение, имеющее доступ к пользовательскому интерфейсу, это приложение – главное, приоритетное. Параллельно с ним может быть запущено фоновое (не имеющее доступа к интерфейсу) приложение, которое получает процессорное время только, когда главное приложение бездействует. Поскольку главное приложение работает во взаимодействии с пользователем, фоновое приложение занимает почти все процессорное время, но оно немедленно прерывается, при появлении событий, требующих активизации главного приложения.

Обработка событий

Интерфейсные приложения PalmOS управляются событиями. Такое приложение представляет собой единственный цикл, каждая итерация которого начинается с получения очередного события. Системный вызов `EvtGetEvent` обеспечивает ожидание и прием события. Полученное событие передается ряду обработчиков в такой последовательности:

- обработчик системных событий;
- обработчик событий меню;
- обработчик событий приложения;

- обработчик диспетчеризации экранных форм;
- обработчик экранных форм.

Если вызванный обработчик распознает событие как подлежащее его обработке, он эту обработку выполняет. В процессе этой обработки он может генерировать и направлять в очередь другие события. Если обработчик выполнил полную обработку события, то он возвращает true, и тогда приложение прерывает цепочку вызовов обработчиков. Алгоритм функционирования приложения, таким образом, выглядит примерно так, как показано на рисунке 7.2.

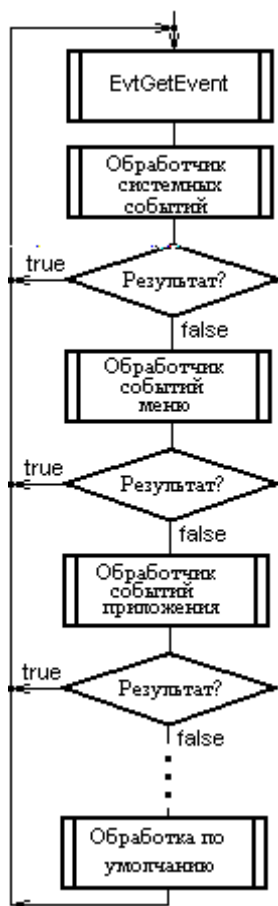


Рисунок 7.2 Обработка событий в приложении PalmOS

Управление энергопотреблением

PDA Palm являются рекордсменами среди устройств такого типа по низкому энергопотреблению, что обеспечивается аппаратными

средствами совместно с ОС. Это достигается наличием в PDA нескольких режимов энергопотребления и их управлением со стороны ОС. Режимы эти следующие:

- Рабочий режим. В рабочем режиме процессор выполняет инструкции. Процессор и вся аппаратура ввода-вывода потребляют энергию в полном объеме. Типичное приложение, переводящее систему в рабочий режим, использует около 5% процессорного времени.
- Ждущий режим. PDA выглядит включенным, процессорные часы активны, но инструкции не выполняются. Процессор работает на пониженном энергопотреблении. Когда процессор получает прерывание, он переходит из ждущего режима в рабочий. Также переключение происходит, когда пользователь начинает вводить информацию световым пером. PalmOS программно переводит устройство в ждущий режим в системном вызове `EvtGetEvent`, если очередь событий пуста. Поступление нового события начнется с прерывания, которое вернет PDA в рабочий режим.
- Спящий режим. PDA выглядит выключенным: дисплей пуст, процессор неактивен, а главные часы остановлены. Активны только часы реального времени и генератор прерываний. PalmOS включает этот режим, когда нет активных действий пользователя в течение некоторого интервала времени или когда пользователь нажимает кнопку выключения. Вход из спящего режима происходит по прерыванию, например, если пользователь нажал на любую кнопку или сработали часы реального времени по заданной программе. Когда система получает одно из этих прерываний в спящем режиме, она переходит в рабочий режим.

Память является одним из наиболее критических ресурсов PDA, и в то же время – наиболее быстро наращиваемым. За время существования PDA Palm и PalmOS доступные объемы памяти в устройстве возросли от 512 Кбайт до 8 Мбайт. Память в устройстве Palm есть оперативная (RAM) и постоянная (ROM). Вся память расположена на карте памяти, на карте может размещаться как ROM, так и RAM-память или обе вместе. Содержимое обоих видов памяти сохраняется даже при "выключении" PDA (переводе его в спящий режим). Архитектура памяти Palm 32-битная. Каждой карте памяти отводится адресное пространство 256 Мбайт. ROM и RAM-память карты разбита на "кучи", размер каждой кучи – не менее 64 Кбайт. Деление памяти на кучи – условное, оно производится ОС и никак не отражается на аппаратной архитектуре памяти. В каждой куче содержится либо ROM, либо RAM-память, но не обе вместе. В RAM-памяти на внутренней карте памяти PDA реализована "динамическая куча". Фактически это и есть оперативная память. В этой куче ОС размещает динамические данные: глобальные переменные, динамические системные области памяти, стек, кучу приложения и сами коды приложений, загружаемых из дополнительных карт. Размер динамической кучи зависит от объема памяти на внутренней карте PDA и от предустановленного программного обеспечения. Каждая куча имеет свой номер-идентификатор. Динамическая куча имеет номер 0. Эта куча инициализируется автоматически всякий раз при рестарте системы. Все другие кучи инициализируются собственными циклами переустановки.

Память распределяется порциями (chunk) переменной длины. Порция располагается в одной куче, выравнивается по границе 2-байтного слова и занимает непрерывную область памяти – от 1 байта до 64 Кбайт. Порции в RAM-памяти бывают динамическими или хранимыми, перемещаемыми или неперемещаемыми. Порции в ROM-памяти бывают только неперемещаемыми и хранимыми.

Управление памятью (прежде всего – динамической, перемещаемой памятью) ведется Менеджером Памяти ОС. Управление хранимой памятью ведется надстройкой над Менеджером Памяти, называемой Менеджером Данных.

Когда Менеджер Памяти выделяет непеременяемую порцию, он возвращает указатель на нее. Этот указатель сохраняет свое начальное значение все время существования порции. Когда Менеджер Памяти выделяет перемещаемую порцию, он возвращает ее манипулятор (handle). Манипулятор является номером элемента в Главной таблице указателей. Главная таблица указателей содержит указатели на порции. Поскольку обращения к перемещаемым порциям происходит через манипуляторы, ОС имеет возможность переместить порцию в памяти и изменить указатель на нее в Главной таблице, манипулятор же порции, которым оперирует приложение, не изменяется. Перемещение, однако, возможно только в те моменты, когда это "позволяет" приложение. Поскольку в системе не предусматривается аппаратное преобразование виртуальных адресов в реальные, для того, чтобы работать с данными памяти, приложение должно иметь указатель на данные. Системный вызов `MemHandleLock()` фиксирует порцию в памяти, то есть запрещает ее перемещение. Этот вызов возвращает указатель на начало порции, через который приложение осуществляет доступ к данным порции. После окончания работы с данными приложение должно снять фиксацию с порции. ОС применяет перемещение порций для борьбы с фрагментацией памяти (внешними дырами). Процедура сжатия памяти вызывается всякий раз при нехватке памяти. Естественно, что чем больше порций будет зафиксировано в памяти, тем менее эффективна будет такая процедура. Поэтому эффективность управления памятью в значительной степени зависит от "грамотного" поведения приложения. В целях снижения

фрагментации ОС выделяет память для перемещаемых порций в начале кучи, а для неподвижных – в конце кучи.

Каждая порция памяти имеет свой локальный (в пределах данной карты памяти) идентификатор. Для неподвижной порции этот идентификатор – ее смещение относительно начала карты, для перемещаемой – смещение относительно начала карты соответствующего ей элемента Главной таблицы указателей. Таким образом, обработка данных не зависит от того, в какой слот будет вставляться карта. Специальный системный вызов превращает номер слота и локальный идентификатор порции в указатель или манипулятор.

Каждая куча начинается с заголовка кучи, который содержит размер кучи и информацию о ее состоянии. Главная таблица указателей располагается сразу вслед за заголовком. Если размера таблицы недостаточно, выделяется ее расширение. Последнее поле таблицы содержит указатель на расширение. Таким образом, может быть выделено сколько угодно расширений, и они связываются в однонаправленный линейный список. Расширения Главной таблицы указателей размещаются в конце кучи. Память, выделяемая для перемещаемых порций, находится сразу за Главной таблицей.

Заголовок кучи не подлежит изменению, поэтому куча может располагаться в ROM-памяти. Поскольку порции в ROM-памяти не могут быть перемещаемыми, Главная таблица кучи в ROM-памяти содержит 0 элементов.

Каждой порции памяти предшествует 8-байтный заголовок, в котором содержится признак свободной/занятой порции, размер порции, счетчик фиксаций и обратная ссылка на Главную таблицу указателей. Свободные участки памяти также имеют формат порций, таким образом, ОС может отследить все распределение памяти, начиная от первой порции и прибавляя к ее адресу размер. Каждая порция может быть зафиксирована

в памяти до 16 раз, каждая новая фиксация просто прибавляет единицу к счетчику фиксаций, а снятие фиксации – вычитает единицу. Порция становится перемещаемой только когда ее счетчик фиксаций обнулится. Для порций хранимой памяти счетчик фиксаций сразу устанавливается в максимальное значение и не уменьшается при попытках снять фиксацию.

Хранимые области памяти в Palm играют ту же роль, что файлы на внешней памяти в других вычислительных системах. Однако в отличие от "традиционных" вычислительных систем, в которых данные для обработки перемещаются в буфер в оперативной памяти, в PalmOS хранимые данные обрабатываются на месте, прямо в постоянной памяти. Работа с хранимыми данными обеспечивается Менеджером Данных, представляющим собой надстройку над Менеджером Памяти.

Данные хранятся в памяти в виде записей. Каждая запись представляет собой порцию памяти. Для выделения, освобождения, изменения размера записи Менеджер Данных обращается к Менеджеру Памяти. Логически связанные записи объединяются в базу данных, база данных является аналогом файла как именованная совокупность данных. Записи базы данных не обязательно располагаются в смежных порциях памяти, они могут быть даже рассредоточены по разным кучам в пределах одной карты памяти.

Каждая база данных имеет заголовок, в котором записано имя базы данных, количество записей в ней и другая управляющая информация. В заголовке же находится и план размещения базы данных, представляющий собой массив дескрипторов записей, входящих в базу. Если весь массив не помещается в заголовке, то последний его элемент содержит указатель (локальный идентификатор) продолжения списка. Каждый дескриптор записи представляет собой 4-байтную структуру, в первом байте которой находятся атрибуты записи (признаки: защиты, удаления, изменения,

занятости), а в оставшихся трех – локальный идентификатор – адрес записи.

API Менеджера Данных представляет собой как бы "гибрид" традиционного файлового API и API памяти. Для того, чтобы работать с базой данных, приложение должно сначала ее "найти" – по символьному имени базы данных определить ее идентификатор (локальный идентификатор заголовка базы данных). Это обеспечивается специальным системным вызовом `DmFindDatabase()`. Затем база данных открывается, при этом в динамической куче создается для нее структура данных – аналог дескриптора открытого файла. С открытой базой данных приложение может работать. Основным системным вызовом доступа к данным – `DmGetRecord()` – возвращает указатель на данные записи с заданным номером, выборка и изменение данных записи производится через этот указатель.

Особый вид базы данных называется ресурсом. Ресурсы служат для хранения специальных данных – программных кодов, изображений, интерфейсных элементов и т.д. Структура ресурса отличается от структуры обычной базы данных только тем, что дескриптор записи ресурса имеет размер 10 байтов, два дополнительных байта описывают свойства ресурса. Еще одна надстройка над Менеджером данных – Менеджер Ресурсов – обеспечивает работу с этой дополнительной информацией. API Менеджера Ресурсов функционально аналогичен API Менеджера Данных.

Кроме того, PalmOS обеспечивает интерфейс файлового потока. При использовании этого API хранимые данные представляются в виде потока байтов, не разделенного на записи. Ограничение 64 Кбайт на размер порции отсутствует. Системные вызовы этого интерфейса (`FileOpen()`, `FileClose()`, `FileRead()`, `FileWrite()`, etc.) аналогичны функциям стандартной библиотеки языка C. Файловый поток является только

интерфейсной надстройкой, с одними и теми же данными можно работать и как с файловым потоком, и как с базой данных.

РАСШИРЕНИЯ И ФАЙЛОВАЯ СИСТЕМА

Начиная с версии 4.0, PalmOS содержит единообразную поддержку новых карт, которые могут расширять возможности PDA. В предыдущих версиях карты, вставляемые в слоты расширения, должны были обязательно соответствовать спецификациям памяти Palm и рассматривались ОС как дополнительная память. В слоты расширения могут вставляться:

- карты RAM-памяти (не обязательно соответствующие спецификациям Palm), содержащие приложения и данные к ним или используемые для специальных целей (например, для создания архивных копий);
- карты ROM-памяти (не обязательно соответствующие спецификациям Palm), содержащие приложения и данные к ним;
- карты ввода-вывода для специальных устройств (например, модема);
- комбинированные карты, содержащие как возможности ввода вывода, так и RAM и ROM-память.

Новая версия ОС рассматривает все эти карты расширения как вторичную память и обеспечивает единообразную работу с ними. Основными компонентами архитектуры расширения PalmOS являются:

- драйверы слота;
- файловые системы;
- Менеджер Виртуальной файловой системы (VFS);
- Менеджер Расширения.

Драйвер слота аналогичен традиционному драйверу устройства, он инкапсулирует детали управления оборудованием карты расширения

данного типа и предоставляет Менеджеру Расширения единый интерфейс для управления картами разных типов. Добавление поддержки нового аппаратного расширения требует включения в системную библиотеку нового драйвера слота.

Файловые системы аналогичны драйверам файловых систем в ОС с инсталлируемыми файловыми системами, они обеспечивают работу с конкретными файловыми системами ПК или других устройств. Обычно в PalmOS предустанавливается файловая система FAT, другие файловые системы могут быть добавлены при необходимости.

Менеджер VFS обеспечивает единый интерфейс системных вызовов (`VFSFileOpen()`, `VFSFileClose()`, `VFSFileRead()`, `VFSFileWrite()`, etc.) для всех файловых систем. Он обеспечивает также возможность работы с памятью на картах расширения с использованием API, подобного тому, который применяется для работы с базами данных в основной памяти.

Наконец, Менеджер Расширения обеспечивает отслеживание вставки/удаления карт расширения и управление драйверами слота.

Взаимодействие с пользователем

Три менеджера в составе PalmOS поддерживают взаимодействие с пользователем "по инициативе системы":

- Менеджер Внимания (attention);
- Менеджер Тревоги (alarm);
- Менеджер Извещения (notification).

Менеджер Внимания отвечает за взаимодействие с пользователем в тех случаях, когда требуется привлечь его внимание. Этот менеджер обеспечивает выдачу сигнала пользователю (звуком, вибрацией, другими специальными эффектами), индикацию события, требующего внимания, на

экране (в виде всплывающего окна или маленького индикатора события), управление со стороны пользователя списком таких событий.

Менеджер Тревоги посылает событие приложению при достижении определенного момента времени. Приложение затем может обратиться к Менеджеру Внимания, чтобы привлечь внимание пользователя.

Менеджер Извещения информирует приложения о наступлении некоторого события. Извещение получают те приложения, которые зарегистрировали свой интерес к данному событию. Приложение затем может обратиться к Менеджеру Внимания.

Пользовательский интерфейс PalmOS позволяет увеличить производительность благодаря уменьшению навигации между окнами, открытию новых диалогов. Размещение структур управления приложением (меню, кнопки и т. д.) упрощено настолько, что пользователь может быстро и эффективно управлять ими. Всего интерфейс обеспечивает 10 основных управляющих структур: формы, диалоги, кнопки, триггеры, переключатели, ползунки, поля, меню, списки, линейки прокрутки. Система также дает возможность разработчикам создавать свои собственные компактные пользовательские интерфейсы размером 160x160 пикселей (размер экрана Palm).

Упрощение навигации достигается также за счет упрощения структуры каталогов VFS PalmOS. Пользователь в большинстве случаев просто не видит этой структуры, он видит на экране иконы доступных для запуска приложений. Эти приложения размещаются в каталоге \PALM\Launcher. Обеспечена автоматическая установка приложения в этом каталоге при добавлении новой карты. Имеется также возможность автоматического запуска приложения при вставке его карты в слот.

Хотя "девизом" PDA Palm и PalmOS является экономия ресурсов, существуют и PDA других фирм, также работающие под управлением PalmOS, которые предоставляют своим владельцам гораздо более широкие

возможности, прежде всего – в части обработки мультимедийной информации (лидером в этом отношении, по-видимому, является фирма Cassio). "Карманная" работа с мультимедийной информацией является объективной тенденцией, и игнорировать ее фирма Palm не сможет. Возможно, в скором времени и облик PalmOS претерпит значительные изменения.

7.3. Операционная система Windows CE

Управление процессами и памятью.

ОС Windows CE [39] рассчитана на значительно большие объемы ресурсов, чем PalmOS, но, соответственно, обеспечивает гораздо больший объем возможностей. Windows CE является членом семейства ОС Windows и в большей части функций обеспечивает общий стандартный API Win32 этого семейства и общий с остальными членами семейства интерфейс пользователя, но в некоторых случаях принятые в Windows CE решения являются специфичными.

Windows CE является многозадачной системой с вытесняющей многозадачностью. Определенные свойства Windows CE дают основание говорить о ней как о системе реального времени. В системе обеспечиваются абсолютные приоритеты, выполняется только процесс (нить) с наивысшим приоритетом. Если два или более процессов имеют высший приоритет, то квант времени (по умолчанию размер кванта – 100 мсек) делится между ними поровну. Всего имеется 256 градаций приоритета, но только 8 самых низших из них возможны для пользовательских процессов, остальные зарезервированы за системными процессами. Windows CE поддерживает вложенные прерывания и "инверсию" приоритетов – повышение приоритета нити, если она захватывает критический ресурс. Windows CE является 32-разрядной

системой и обеспечивает в основном тот же API Win32, что и другие ОС Windows. Ядро Windows CE может адресовать до 256 Мбайт физической памяти, но в виртуальном адресном пространстве объем которого – 4 Гбайт, физическая память отображается в младшие 2 Гбайт, как показано на рисунке 7.3.

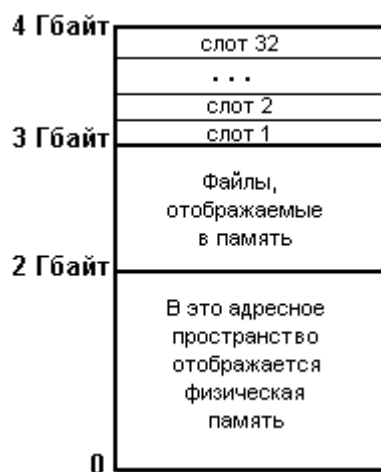


Рисунок 7.3 Виртуальное адресное пространство Windows CE

В старшей части памяти адресное пространство от 2 до 3 Гбайт отводится для совместно используемой памяти (в терминологии Windows – файлов, отображаемых в память), а пространство от 3 до 4 Гбайт делится на "слоты" с номерами от 1 до 32, каждый из которых представляет адресное пространство одного из процессов. Таким образом, в системе может выполняться одновременно до 32 процессов, частное адресное пространство каждого процесса – 32 Мбайт (не считая файлов, отображаемых в память). "Слот" 0 отображается на физическую память, он отдается активному в текущий момент процессу. В адресном пространстве каждого процесса помимо кодов процесса создаются области памяти для статических данных, куча и стек для каждой нити процесса. Для статических данных выделяются отдельные области памяти – для изменяемых и для неизменяемых данных. Для каждого процесса создается куча по умолчанию (384 страницы по 1 Кбайт), но процесс может

создавать новые кучи в пределах своего адресного пространства. Выделенные в куче блоки памяти не перемещаются, что может приводить к фрагментации памяти в куче. Размер стека для нити – 1 Мбайт, и он не может быть изменен. Количество нитей в процессе ограничено только возможностью выделения памяти для стеков нитей. Память для стека выделяется по мере необходимости, постранично. Если для растущего стека нити не хватает страниц памяти, нить блокируется.

Программы, выполняемые в Windows CE, могут находиться в RAM-или в ROM-памяти. Если программа находится в ROM-памяти, но не содержит изменяемых данных, она выполняется "на месте". Если же программа содержит изменяемые данные, она для выполнения копируется в RAM-память. Копирование происходит постранично, по требованию.

Общие области памяти, называемые в Windows файлами, отображаемыми в память, в адресное пространство процесса не входят. Они могут использоваться для получения процессом дополнительной памяти сверх лимита 32 Мбайт.

Управление внешними данными

Управление внешними данными в Windows CE основывается на концепции "хранилища объектов" (object store). Хранилище объектов играет ту же роль, что и дисковая память в настольных вычислительных системах: оно обеспечивает постоянную память для хранения приложений и их сохраняемых данных. Хранилища объектов могут быть трех типов: файловые системы, базы данных и реестры (registry), причем все они могут разделять одну и ту же физическую память. Однако, первые два типа могут также размещаться и в ROM-памяти, на внешних устройствах или в отдельных системах, реестры же – только в RAM-памяти. Объектами, находящимися в хранилище, могут быть:

- ключ реестра;

- значение реестра;
- файл (метаинформация файла);
- порция (chunk) данных файла (размер порции – 4 Кбайт);
- запись базы данных (до 4 Кбайт);
- расширение записи базы данных (до 4 Кбайт);
- база данных (метаинформация базы данных).

Каждый объект имеет уникальный (в пределах тома) идентификатор, который используется для доступа к объектам.

Windows CE работает с тремя типами файловых систем: файловая система в ROM-памяти, файловая система в RAM-памяти и файловая система FAT на внешних устройствах, картах расширения памяти и PC Card. Разработчики могут создавать и регистрировать и другие файловые системы. Независимо от того, на каком физическом типе памяти располагается файловая система, работа с нею выполняется через стандартный файловый API Win32. Для упрощения операций с памятью Windows CE не применяет концепцию текущего каталога, но все ссылки на объекты содержат полный маршрут.

База данных в Windows CE представляет собой нечто, являющееся упрощенным вариантом СУБД. API баз данных в Windows CE оригинальный. База данных состоит из записей. Каждая запись состоит из полей (свойств). Запись может состоять из переменного количества полей, память выделяется только под реально существующие поля. Каждое поле предваряется 4-байтным заголовком, в котором содержится идентификатор поля и код типа данных в поле. Каждая запись предваряется 20-байтным заголовком, содержащим метаданные записи. Вся база данных имеет символьное имя (до 32 символов) и тип (целое число).

Для базы данных может быть создано до 4 индексов быстрого поиска – каждый по значения какого-либо одного поля. При открытии базы

данных (системный вызов `CeOpenDatabaseEx`) может быть указан один из этих четырех индексов, и в течение этого сеанса работы с базой данных используется только индекс, заданный при открытии. Прежде, чем читать или писать запись базы данных, ее следует найти. Системный вызов `CeSeekDatabase` ищет в базе данных запись, поиск может задаваться: по абсолютному или относительному значению поля, по абсолютному или относительному номеру в заданном при открытии базы данных индексе, по идентификатору объекта-записи. Если запись найдена, указатель поиска устанавливается на эту запись. Последующая операция чтения или записи оперируют с той записью, на которую установлен указатель поиска. Работа с содержимым базы данных выполняется при помощи системных вызовов `CeReadRecordPropsEx` и `CeWriteRecordProps`. Эти вызовы позволяют соответственно читать поля записи или записывать поля в запись. Для определения того, с какими именно полями работает системный вызов, должен быть определен массив идентификаторов полей. Чтение значений полей записи может выполняться не только в локальную кучу программы, но и в любую доступную программе область памяти.

Реестры Windows CE хранят конфигурационные установки: данные о приложениях, драйверах, настройки пользователей и т.п. Реестры организованы в иерархическую структуру с ключами и значениями. Ключ может содержать значение или другие ключи – в этом ключ подобен каталогу файловой системы. В иерархической структуре ключей возможно не более 16 уровней. Windows CE поддерживает три "корневых" ключа, в которые записываются другие ключи, задающие параметры соответствующего типа:

- `HKEY_LOCAL_MACHINE` – данные о конфигурации аппаратуры и о драйверах;

- HKEY_CURRENT_USER – конфигурационные данные пользователя;
- HKEY_CLASSES_ROOT – конфигурационные данные приложений.

Ограничение на длину ключа – 255 символов, ограничение на размер значения – 4 Кбайт. Для работы с реестрами используется API Win32.

Windows CE продолжает развиваться, но наряду с этой ОС фирма Microsoft предлагает также Windows NT Embedded. Последняя обеспечивает примерно ту же функциональность, но ориентированную не на карманные ПК, а на вычислительные устройства, встраиваемые в различную аппаратуру и строится на ядре Windows NT. Эта технология развивается в новой версии ОС для встроенных применений – Windows XP Embedded. Эта ОС строится на базе ядра Windows 2000 (Windows NT 5) и обеспечивает функциональность, аналогичную Windows CE и Windows NT Embedded. Основное нововведение в Windows XP Embedded – развитая библиотека компонентов и средства разработки приложений.

7.4. Новые тенденции встроенных ОС

Ограниченность ресурсов тонких клиентов определила то обстоятельство, что ОС таких устройств, во-первых, чрезвычайно экономны в потреблении ресурсов, во-вторых, являются ОС реального времени. Основным признаком ОС реального времени является иерархия приоритетов прерываний и возможность приостанова обработки текущего прерывания при поступлении прерывания с более высоким приоритетом. Поскольку прерывания сигнализируют о внешних событиях, указанное свойство позволяет ОС реального времени своевременно реагировать на такие события. И PalmOS, и Windows CE являются ОС реального времени.

Большинство других ОС, используемых как встроенные, также подходят под это определение (например, QNX). Однако, развитие аппаратных средств – увеличение быстродействия, разрядности, объемов памяти приводит к тому, что в качестве встроенных начинают использоваться "облегченные" версии стандартных ОС [33]. При высоком быстродействии аппаратуры эти ОС позволяют обеспечить приемлемое время реакции даже не для приоритетных прерываний. Использование большей ОС и большего количества памяти, чем требует ОС реального времени, обходится дороже, но, во-первых, позволяет добавлять в системы больше возможностей, а во-вторых, позволяет сократить сроки разработки за счет применения инструментальных средств стандартных ОС и привлечения разработчиков, не обладающих специфическим опытом разработок для ОС реального времени.

Например, технология Microsoft Windows NT Embedded для встроенных ОС развивается в новой версии ОС для встроенных применений – Windows XP Embedded. Эта ОС строится на базе ядра Windows 2000 (Windows NT 5) и обеспечивает функциональность, аналогичную Windows CE и Windows NT Embedded. Основное нововведение в Windows XP Embedded – развитая библиотека компонентов и средства разработки приложений. Хотя по некоторым сообщениям от фирмы Microsoft Windows XP Embedded должна быть основным предложением фирмы для мобильных и встроенных систем, еще до конца 2002 года фирма планирует выпустить ОС Windows CE .Net.

Заметной фигурой на рынке встроенных ОС стала также ОС Linux. ОС Linux модульная в своей основе. Ядро (фундаментальные элементы ОС, такие как управление памятью и файлами) занимает примерно один 1 Мбайт. Оно может быть легко выделено и дополнено модулями, подходящими для встроенных систем. Linux как ОС Открытого Кода предоставляет множество привлекательных возможностей для

производителей и разработчиков и, конечно же, рассматривается ими как альтернатива Microsoft.

Таким образом, можно указать на две тенденции в развитии встроенных ОС. Одна, представляемая PalmOS, характеризуется прежде всего экономичностью и некоторым аскетизмом в возможностях, другая, представляемая, например, Windows XP, – более затратным отношением к ресурсам, но и большими возможностями для пользователя, выражающимися прежде всего в широком использовании мультимедийных средств. Естественно, развитие аппаратных средств делает более перспективной вторую тенденцию. Очевидно, что фирма Palm также не собирается оставаться в стороне от общего потока, о чем свидетельствует, например, приобретение ею в 2001 г. технологии BeOS, известной своими мультимедийными возможностями, апробированными также и во встроенных применениях. Однако, как бы интенсивно не развивались аппаратные ресурсы, всегда на рынке будет сохраняться устойчивый спрос на более дешевые и более экономичные решения, таким образом, опыт и технологии PalmOS не останутся невостребованными.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем состоит специфика ОС для тонких клиентов?
2. Обеспечивает ли многозадачность ядро PalmOS? Обеспечивает ли многозадачность PalmOS?
3. Как в PalmOS обрабатываются события? Какие компоненты ОС управляют обработкой внешних событий?
4. Какие события заставляют PalmOS менять режим энергопотребления?
5. Какую роль играет в PalmOS динамическая куча?

6. Что такое перемещаемая и неперемещаемая память? В чем состоит различие в управлении тем и другим видом памяти в PalmOS?
7. Какие интерфейсы доступа к хранимым данным обеспечивает PalmOS?
8. Сопоставьте многозадачность Windows 9x и Windows CE. Какая из этих ОС является "более многозадачной"?
9. Сколько процессов может одновременно выполняться в Windows CE? С чем связано ограничение на их максимальное количество?
10. Опишите особенности управления памятью в Windows CE.
11. Какие объекты могут находиться в хранилище объектов Windows CE? К каким основным трем типам эти объекты сводятся?
12. Что такое база данных в Windows CE?
13. Является ли ОС реального времени Windows CE? Windows NT Embedded? Windows XP Embedded?
14. К каким изменениям в ОС может привести применение в PDA процессоров, поддерживающих динамическую трансляцию адреса?
15. В связи с увеличением вычислительной мощности тонких клиентов, можно ли считать "экономную" концепцию PalmOS неперспективной?

Глава 8. Операционные системы фирмы Apple

8.1. Фирма Apple и компьютеры Macintosh

История Apple является хрестоматийным примером того, как талант и предприимчивость приносят успех. Персональный компьютер, собранный в гараже Стивом Возняком и Стивом Джобсом послужил основой для создания в 1976 г. фирмы Apple Computer Company и компьютера Macintosh. Хотя более чем 25-летняя история фирмы далеко

не безоблачна, она переживала взлеты и падения, можно утверждать, что во все периоды своей деятельности фирма имела собственную концепцию персональных вычислений и продукция Apple по качеству и по функциональным возможностям опережала конкурентов. Фирма выжила в конкурентной борьбе как с супергигантами компьютерной индустрии, так и с конкурентами, чей стиль борьбы на рынке никак нельзя назвать честным, и в настоящее время компьютеры Apple являются единственной реальной альтернативой платформе Wintel в настольных вычислениях.

Хотя компьютеры Apple являются компьютерами универсального назначения, их аппаратное и программное оснащение средствами обработки мультимедийной информации всегда было несколько избыточным для "рядового" покупателя. Конечно, это утверждение можно оспаривать, но то, что большинство таких пользователей предпочли более дешевую платформу Wintel – очевидный факт. Как и то, что пользователи, профессионально работающие в области дизайна, издательской деятельности и в других областях, связанных с обработкой мультимедийной информации, предпочитают Macintosh. Следует отметить, что при более высокой стартовой цене компьютеры Apple в итоге обходятся своим покупателям дешевле, так как дольше сохраняют свою пригодность, и их пользователи избавлены от того беличьего колеса непрерывных модернизаций, в котором вынуждены крутиться пользователи Wintel.

Компьютеры Macintosh первоначально базировались на процессорах Motorola 680x0 (M68K), внедряя новые поколения этих процессоров. В 1992 г. в результате совместного проекта фирм Apple, IBM и Motorola был разработан процессор PowerPC, и семейство компьютеров Macintosh разделилось на две ветви – одна на процессорах M68K, другая – процессорах PowerPC, также выпускаемых фирмой Motorola. В настоящее время все новые модели Macintosh базируются на PowerPC.

Программное обеспечение Macintosh, как и аппаратное, в значительной степени ориентировано на предоставление пользователям развитых мультимедийных возможностей. Графический пользовательский интерфейс компьютеров Apple служит образцом для всех других систем. Операционная система Mac OS эволюционировала на протяжении долгих лет и лишь в 1998 г. уступила место ОС Mac OS X.

8.2. Mac OS

Хотя в оригинальной документации нам не удалось найти определения архитектуры MacOS [28], мы возьмем на себя смелость определить ее как модульно-иерархическую, как представлено на рисунок 8.1.

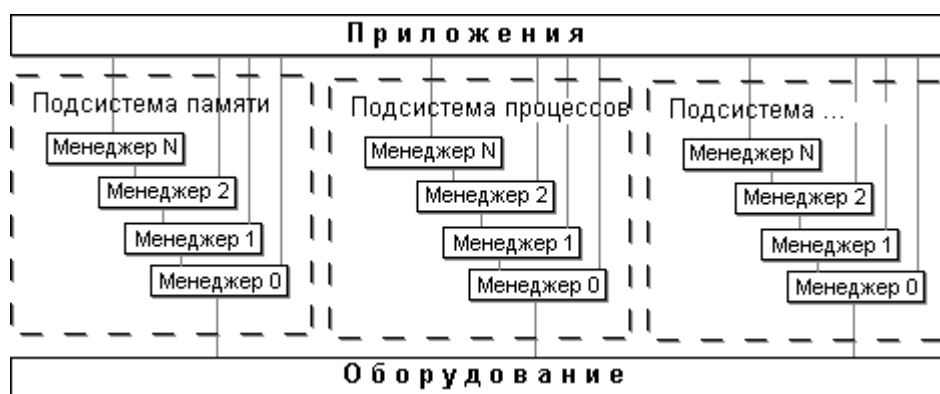


Рисунок 8.1 Архитектура Mac OS

Систему, по-видимому, можно разделить на несколько подсистем, каждая из которых выполняет управление определенным видом ресурсов (памятью, задачами, файлами, средствами коммуникаций и т.д.). Подсистема состоит из нескольких Менеджеров, каждый из которых обеспечивает более высокий уровень абстракции ресурсов. Менеджеры более высокого уровня используют средства Менеджеров низкого уровня

своей подсистемы, а также и других подсистем. API же системы предоставляет доступа к возможностям практически любого уровня абстракции. На рисунке 8.2, например, показана структура подсистемы управления взаимодействием процессов.

Управление памятью

При работе с реальной памятью Mac OS обеспечивает работу с адресным пространством размером 16 Мбайт (24-разрядный адрес). Разумеется, все адресное пространство не обязательно поддерживается реальной памятью, заполнение адресного пространства реальной памятью может быть фрагментировано. До 8 Мбайт в верхней части адресного пространства составляет пространство ввода-вывода.

Память в такой модели выделяется разделами. Нижняя часть памяти (от адреса 0) составляет системный раздел. В нем размещены глобальные переменные системы и системная куча. В системной куче выделяется память для буферов, системных структур данных и системных кодовых сегментов.

Каждому приложению выделяется раздел приложения. В разделе приложения содержится:

- управляющая информация приложения, так называемый "мир A5" (A5 world);
- стек приложения;
- куча приложения.

"Мир A5" (название происходит от имени регистра микропроцессора M 68K, который используется для адресации) содержит:

- глобальные переменные приложения;
- глобальные переменные QuickDraw (подсистемы экранного отображения);

- параметры приложения;
- таблицу переходов.

Стек приложения используется для сохранения адресов возврата и выделения памяти для локальных переменных. В куче размещаются коды и данные приложения. Кроме того, приложению могут выделяться по запросу блоки памяти вне его раздела.

Память в куче выделяется блоками переменной длины. Блоки могут быть перемещаемыми или неперемещаемыми. Обращение к неперемещаемому блоку производится по прямому адресу. Обращение к перемещаемому блоку производится с применением косвенной адресации через так называемый главный блок указателей (master pointer block). Для каждого приложения система создает такой блок определенного по умолчанию размера, размер блока может быть увеличен самим приложением. Такой способ выделения памяти приводит к образованию "внешних дыр", которые могут уменьшать объем доступной для приложения памяти. Для борьбы с этим явлением система производит (при нехватке памяти) дефрагментацию кучи – переписывает в памяти все перемещаемые блоки таким образом, чтобы внешние дыры слились в одну свободную область в верхней части кучи. При переносе блоков корректируется главный блок указателей, таким образом, перенос остается прозрачным для приложения. Наличие в куче неперемещаемых блоков снижает эффективность сжатия кучи, поэтому система стремится разместить все перемещаемые блоки в нижней части кучи. Если при размещении перемещаемого блока оказывается, что то место в нижней части кучи, на которое он претендует, занято перемещаемым блоком, система переносит перемещаемый блок в другое место и освобождает место для неперемещаемого.

Перемещаемый блок может также быть объявлен удаляемым: системе разрешается удалять его при нехватке памяти в куче приложения.

Работая с удаляемым блоком, программист должен всякий раз, начиная работу с ним проверить его наличие в памяти и отменить признак, разрешающий удаление.

Введение в компьютеры фирмы Apple динамической трансляции адресов позволило перейти к 32-разрядному размеру адреса и, таким образом, обеспечивать виртуальное адресное пространство размером в 4 Гбайт. Динамическая трансляция адресов использует страничную модель виртуальной памяти для расширения адресного пространства. Страничный обмен использует алгоритм LRU. Структура нижней части адресного пространства (до границы 16 Мбайт) – такая же, как и в 24-разрядной модели, что обеспечивает прозрачное выполнение 24-разрядных приложений в новой среде. Для 32-разрядных приложений могут выделяться дополнительные разделы выше 16-Мбайтной границы.

В описанной выше модели реальной памяти, расширенной затем за счет динамической трансляции адресов, сложилась сегментная архитектура выполнения приложений, которую называют "классической" архитектурой 68К. Приложение в этой архитектуре состоит из сегментов размером до 32 Кбайт каждый. Сегментная архитектура поддерживается Менеджером Сегментов в составе Mac OS. Для каждого приложения автоматически создается и загружается при запуске Сегмент 0, остальные сегменты загружаются по требованию.

Связь между сегментами обеспечивается через таблицу переходов (jump table), которая размещается вместе с "миром A5". Таблица переходов содержит адреса входных точек в сегментах, таким образом, обращения к процедурам в других сегментах производятся через таблицу переходов. Сегменты размещаются в перемещаемых блоках памяти в куче приложения и, таким образом, могут быть перемещены в памяти с коррекцией содержимого таблицы переходов. Загрузка сегментов производится автоматически при первом обращении к любой входной

точке сегмента. Сегмент может быть также и выгружен из памяти, но это приложение должно сделать явным образом: выполнить системный вызов, помечающий сегмент как удаляемый. Помеченный таким образом сегментный блок может быть удален из памяти при нехватке памяти.

Архитектура CFM

В новых версиях Mac OS на M68K и PowerPC введена иная архитектура выполнения приложений. Она поддерживается Менеджером Кодовых Фрагментов – CFM (Code Fragments Manager) в составе ОС, поэтому называется архитектурой CFM. Архитектура CFM в максимальной степени использует концепцию динамической компоновки. Программа (кодовый ресурс) в терминах Mac OS состоит из двух ответвлений (fork) – ресурса (кодов) и данных (статических данных). Приложение в архитектуре CFM загружается системным загрузчиком (Finder). Другой вид кодовых ресурсов составляют разделяемые библиотеки и подключения (plug-in). Эти ресурсы CFM подключает к приложению во время выполнения и обеспечивает возможность совместного использования их разными приложениями. При этом они не используют память приложения, а записываются отдельно. Разница между библиотеками и дополнениями состоит в том, что первые подключаются автоматически, а вторые – специальным системным вызовом. Подключения могут содержать собственную главную процедуру (функцию main). Установленная связь процесса с фрагментом называется соединением (connection). Приложение может иметь соединения с несколькими фрагментами, также и фрагмент может быть соединен с несколькими приложениями одновременно. При обработке фрагментов CFM использует концепцию "застежек" (closure). Застежка является набором соединений процесса с фрагментами. Застежка представляет собой "корневой фрагмент", к которому CFM обращается для выполнения

связывания с любыми разделяемыми библиотеками. Как правило, процесс имеет одну застёжку, но для связывания с подключением (plug-in) создается отдельная застёжка подключения.

Как упоминалось выше, для фрагмента может быть создано несколько соединений – как с одним процессом, так и с несколькими. При этом ответвление кода и ответвление данных соединяются отдельно и не обязательно в одной застёжке. Если для кодового ответвления создается несколько соединений, то все соединения совместно используют один экземпляр кода. Для ответвления данных возможна глобальная реализация (один и тот же экземпляр данных используется всеми соединениями в системе) или реализация для процесса (CFM создает отдельную копию данных для каждого процесса). Возможность глобальной реализации или реализации для процесса определяется при создании разделяемых библиотек. Установление связи между процессом и библиотечным фрагментом осуществляется при помощи таблицы связываний, помещаемой редактором связей в каждое приложение. В этой таблице предусматриваются строки для всех библиотечных входных точек, к которым обращается приложение. В кодах приложения обращения к внешним точкам имеют вид косвенных обращений к таблице связываний. При создании соединения CFM находит нужную библиотеку и заполняет таблицу связываний приложения ссылками на адреса входных точек в оглавлении библиотеки.

Динамическая компоновка является средством, обеспечивающим возможность модификации разделяемых (например, системных) библиотек прозрачно для приложения. Для обеспечения совместимости версий приложений и библиотек в таблицу связываний приложения включаются граничные номера версий библиотеки, с которыми приложение может работать. CFM при создании соединения выполняет сопоставление этой информации с версией найденной библиотеки.

Управление процессами и нитями

Mac OS во всех своих версиях являлась системой с кооперативной многозадачностью. Процессом в системе является запущенное приложение или, в некоторых случаях, открытый аксессуар рабочего стола. В каждый момент только один процесс находится на переднем плане – тот, с которым взаимодействует пользователь, остальные являются фоновыми. Возможны также только-фоновые процессы, разработанные без пользовательского интерфейса. Переключение процессов происходит только в том случае, если процесс переднего плана приостанавливается, если он выдает системный вызов `WaitNextEvent` или `EventAvail`, но в системной очереди событий нет для него сообщений. Только при выполнении этих системных вызовов возможно переключение контекста. Различается переключение контекста значительное (`major`) и незначительное (`minor`). Первое происходит в том случае, если фоновый процесс становится процессом переднего плана. В этом случае ОС посылает процессу переднего плана "событие приостанова". Обработывая это событие, процесс переднего плана может выполнить требуемые прикладные операции, связанные с переходом на задний план. Когда процесс переднего плана в следующий раз выдаст системный вызов `WaitNextEvent` или `EventAvail`, он будет задержан. Фоновый процесс в этом случае получает от ОС "событие возобновления". Незначительное переключение происходит, когда фоновый процесс получает процессорное обслуживание, не переходя на передний план, например, когда процесс переднего плана ожидает действий пользователя. В этом случае, когда происходит событие, которого ожидает процесс переднего плана, фоновый процесс приостанавливается при следующей выдаче им системного вызова `WaitNextEvent` или `EventAvail`,

независимо от того есть ли для него сообщения в системной очереди событий.

Другой формой, использующей процессорный ресурс, являются в Mac OS задачи (task). Задачи представляют собой обработчики прерываний. Приложение имеет возможность установить собственную обработку некоторых прерываний. Пользовательская обработка прерываний поддерживается:

- Менеджером Времени, который позволяет приложению выполнять задачи через заданные интервалы времени;
- Менеджером Регенерации (Vertical Retrace Task), который позволяет выполнять задачи между циклами восстановления изображения на экране;
- Менеджером Уведомления (Notification Manager), который обеспечивает как для процессов, так и для задач авральную сигнализацию пользователю (например, в случае ошибки);
- Менеджером Устройств, который дает возможность драйверам обрабатывать прерывания от устройств.

Все эти менеджеры используют установленную приложениями информацию о задачах, помещаемую в системные очереди.

Когда задача получает управление, она не обязательно выполняется в контексте приложения, создавшего данный обработчик прерывания. Поэтому действия, выполняемые в составе задачи, существенно ограничены, и для установления обработчика прерывания с создавшим его приложением должны быть выполнены определенные специальные действия. Некоторые обработчики прерываний не деинсталлируются автоматически при завершении установивших их приложений.

Mac OS обеспечивает также механизм нитей. Нить, как и в большинстве других систем, имеет собственный вектор состояния процессора и собственный стек. Нити могут создаваться по отдельности

или же может быть сначала создан пул нитей, а затем новые нити создаются из пула. Второй вариант обеспечивает более рациональное использование ресурсов, в частности, меньшую фрагментацию памяти. В первой реализации Менеджера Нитей поддерживалась вытесняющая многопоточность в рамках невытесняющей многозадачности, но затем дисциплину управления нитями сделали также кооперативной. Нить, получившая процессорное обслуживание, может отдать его либо уступив процессор любой нити (в этом случае готовые к выполнению нити выбираются из очереди по дисциплине FCFS), либо уступив процессор какой-то конкретной нити, либо просто изменив свое состояние на ожидание. Отличие последнего случая от первых двух состоит в том, что выполнение нити приостанавливается даже если нет других готовых нитей. Пользователь имеет возможность также определить свою процедуру диспетчеризации нитей, которая выполняется перед выполнением системного планировщика.

Хотя при невытесняющей многозадачности в этом нет необходимости, предусмотрены специальные системные вызовы – скобки критической секции. Внутри критической секции просто игнорируются выдаваемые нитью системные вызовы, уступающие процессор.

Интересной особенностью Mac OS является возможность создания также пользовательских процедур переключения контекста для нитей. Для каждой нити может быть создано две таких процедуры, одна из них выполняется перед переключением контекста на другую нить, вторая – при переключении контекста на данную нить.

Поскольку MasOS работает на двух разных аппаратных платформах, имеются некоторые различия в форматах структур данных для платформ M68K и PowerPC (например, в PowerPC роль регистра A5 играет другой регистр), но эти различия не касаются качественной структуры и алгоритмов обработки. Системы программирования для Mac OS позволяют

создавать программы в кодах той или другой платформы, системой поддерживаются также программные ресурсы "толстого" (fat) формата, содержащие код программы для обеих платформ и, следовательно, переносимые без каких-либо дополнительных действий.

Средства взаимодействия

Mac OS предусматривает весьма развитые механизмы взаимодействия между процессами, основанные прежде всего на очередях сообщений (в Apple они называются событиями) и одинаково применимые для локального и для удаленного взаимодействия. Эти механизмы обеспечиваются набором менеджеров в составе ОС, которые выстраиваются в иерархическую структуру: менеджер более высокого уровня пользуется услугами менеджеров нижних уровней (см. рисунок 8.2).

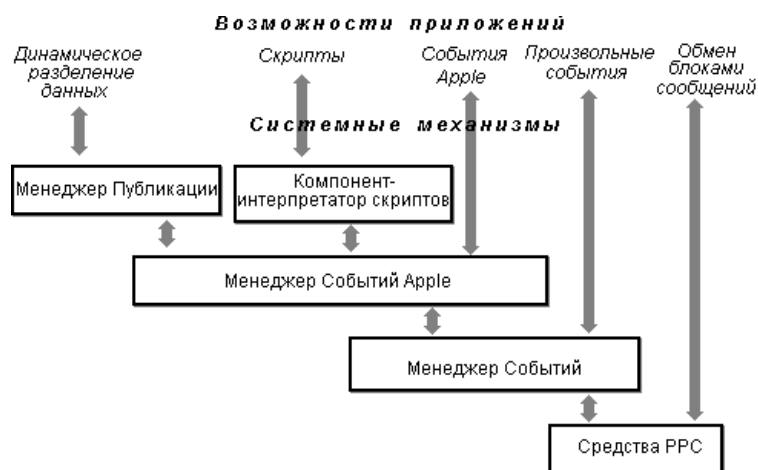


Рисунок 8.2 Средства взаимодействия приложений в Mac OS

Приложение может пользоваться услугами менеджеров любого уровня иерархии, что обеспечивает широкий спектр возможностей взаимодействия, включающий в себя:

- Динамическое разделение данных, обеспечиваемое Менеджером Публикации (Edition Manager). Позволяет приложению копировать данные из одного документа (издателя) в другой документ (подписчик) с автоматическим обновлением данных у подписчика, если они изменены у издателя.
- Обмен "событиями Apple". События Apple – высокоуровневые события, которые соответствуют протоколу AEIMP (Apple Event Interprocess Messaging Protocol), обеспечиваемому Менеджером Событий Apple. Средства Менеджера Событий Apple позволяют приложению-клиенту формировать и посылать события Apple, представляющие собой обычно некоторый запрос на обслуживание, а приложению-серверу – принимать события и отвечать на них. Имеется несколько стандартных комплектов событий Apple (обязательный комплект, комплект ядра, комплекты текста и базы данных), применение которых обеспечивает взаимодействие несвязанных приложений. Наряду с событиями стандартных комплектов пользователями могут вводиться собственные события Apple, интерпретируемые самими приложениями.
- Обмен другими событиями. Менеджер Событий обеспечивает для приложений обмен событиями, не соответствующими протоколу AEIMP.
- Обмен блоками сообщений. Средства PPC (Program-to-Program Communications) позволяют приложениям обмениваться большими блоками данных на низком уровне управления. При применении этих средств приложения, участвующие в диалоге, должны выполняться одновременно и обмениваться данными через общий коммуникационный порт.

Отдельно следует назвать скрипты, функциональность которых выходит за рамки только взаимодействия между процессами. Скрипт представляет собой фактически программу, написанную на интерпретирующем языке высокого уровня AppleScript. Скрипты могут быть записаны в приложения, в документы или в отдельные файлы, загружаемые системным загрузчиком Finder. С точки зрения использования скриптов различаются следующие свойства приложений.

- Приложения, выполняющие скрипты (scriptable). Такое приложение способно реагировать на стандартные события Apple, посылаемые интерпретатором AppleScript, и, следовательно, выполнять действия, определенные в скрипте. При посылке сообщения такому приложению интерпретатор использует специальный ресурс типа “aete” (Apple event terminology extension), в котором описывается соответствие высокоуровневого описания терминов, используемых в скрипте, кодам событий Apple, которые обрабатывает приложение.
- Регистрирующие приложения (recordable). Такое приложение посылает события самому себе. Обычно через события обеспечивается связь вычислительной части приложения с частью, обеспечивающей пользовательский интерфейс. Параллельно с передачей события между частями регистрирующего приложения копия события поступает интерпретатору, который, пользуясь ресурсом типа “aete”, записывает действия, определяемые данным событием в виде скрипта.
- Приложения, выполняющие скрипты и манипулирующие ими. Приложение может записывать и выполнять скрипты,

устанавливая соединение с интерпретатором языка скриптов как с подключением (plug-in).

- В одном приложении могут объединяться все три описанные свойства.

Мы говорили о языке AppleScript, однако на самом деле механизм скриптов более гибок. Он строится по принципу Открытой Архитектуры Скриптов OSA (Open Scripts Architecture). OSA допускает использование любых языков для написания скриптов. Интерпретатор того или иного языка скриптов является выбираемым компонентом. Менеджер Компонентов выбирает заданный компонент для выполнения того или иного скрипта. Интерпретатор AppleScript, таким образом, является лишь одним из возможных компонентов в системе OSA.

Ввод-вывод и файловая система

Корневой структурой в управлении вводом-выводом является Таблица Устройств, неподвижная в системной куче. Каждый элемент Таблицы Устройств адресует один Блок Управления Драйвером. Блок Управления Драйвером является перемещаемым в системной куче и содержит адрес тела драйвера и адрес начала очереди запросов к драйверу. Драйверы бывают синхронные и асинхронные. Первые обслуживают обычно символьные устройства и полностью завершают транзакцию ввода-вывода до возвращения управления. Вторые применяются для блочных устройств и только иницируют операцию ввода-вывода; эти драйверы используют прерывания от устройств для того, чтобы вновь получить управление и завершить транзакцию.

Запросы на ввод-вывод имеют стандартную форму и выстраиваются в очереди к устройствам. Очереди управляются Менеджером Устройств по дисциплине FCFS. Различают запросы асинхронные, синхронные и неотложные. Асинхронный запрос просто ставится в очередь, после чего

управление возвращается выдавшему его приложению. Синхронный запрос переводит приложение в ожидание до выполнения запроса. (Синхронный/асинхронный тип запроса не связан с синхронным/асинхронным типом драйвера.) Неотложный запрос передается Менеджером Устройств прямо на драйвер в обход очереди. Поскольку это может произойти в тот момент, когда драйвер обрабатывает другой запрос, драйвер, которому могут посылаться неотложные запросы, должен быть реентерабельным.

Все операции, выполняемые драйвером, сводятся к нескольким видам, и каждый драйвер должен иметь входные точки в процедуры обработки следующих видов запросов:

- открытие – выделение памяти и инициализация устройства;
- закрытие – деактивизация драйвера и устройства;
- управление – выполнение специфических для устройства функций
- статус – получение информации от драйвера, эта процедура специфична для устройства и может быть необязательной;
- базисные (prime) операции – ввод и вывод, эта процедура также необязательна и может обеспечивать либо ввод, либо вывод, либо и то и другое.

Файловые системы Mac OS называются Иерархическими Файловыми Системами – HFS (Hierarchical File System) и HFS Plus. Как и файл программы, любой файл состоит из двух ответвлений (fork) – ресурсов и данных. В частном случае одно из ответвлений может быть пустым. Ответвление данных не структурировано, ответвление ресурсов содержит карту ресурсов и сами ресурсы, файл может иметь до 2700 ресурсов. Если, например, файл является файлом приложения, ресурсы описывают меню и диалоговые окна приложения, его иконки, события и т.д. и содержат

исполняемый код приложения; если файл является, например, документом, ответвление ресурсов содержит размещение окна документа, иконки, шрифты и т.д. Строго говоря, граница между ресурсами и данными не очень четкая. Информация файла может быть помещена как в ответвление данных, так и в ответвление ресурсов. В ответвление ресурсов помещаются те данные, которые ограничены по размеру и количеству значений.

Дисковое пространство состоит из секторов размером по 512 байт каждый, но распределение дисковой памяти ведется кластерами (в Mac OS они называются блоками распределения). Блок распределения содержит целое число смежных секторов. Размер блока распределения фиксирован для данного тома.

Каждый том физической файловой системы HFS имеет заголовок тома, содержащий общую информацию о томе, такую, как: дата создания и общий размер тома, число файлов на томе, а также расположение остальных управляющих структур файловой системы. Заголовок всегда располагается в секторе 2, копия заголовка размещается в конце тома.

HPFS Plus для управления размещением информации на томе использует специальные файлы, которые размещаются не на фиксированных позициях в томе, а в произвольных местах пространства данных. Различаются следующие специальные файлы.

Файл каталога – содержит информацию об иерархии папок и файлов на томе. Каталог организован как В-дерево и содержит записи четырех видов:

- запись папки – информация об отдельной папке
- запись файла – информация об отдельном файле;
- запись связи папки – информация о родительской папке для данной папки;

- запись связи файла – информация о родительской папке для данного файла.

Информация о файле включает в себя два плана размещения – для ответвления ресурса и для ответвления данных. Размещение файла выполняется экстендами переменной длины, часть плана, размещаемая в записи файла, содержит массив из дескрипторов 8 первых экстентов. Если файл фрагментирован на большее число экстентов, дескрипторы дополнительных экстентов находятся в файле переполнения экстентов.

Файл переполнения экстентов – содержит информацию о дополнительных экстендах файлов, не поместившихся в основные записи файлов в каталоге. Этот файл общий для всего тома и также организован как В-дерево, ключами в котором являются: идентификатор файла, тип ответвления и адрес начала экстента относительно начала файла.

Файл атрибутов – структура, введенная для будущих реализаций, предусматривающих наличие именованных ответвлений в файле. Организован как В-дерево.

Файл размещения – представляет собой битовую карту свободных/занятых блоков распределения. Файл размещения применяется только в HFS Plus, в HFS его функцию выполняла отдельная "область битовой карты", размещавшаяся на томе по фиксированному адресу.

Пусковой файл – файл, содержащий информацию для загрузки с диска HFS операционной системы, отличной от Mac OS.

Интерфейс пользователя

Фирма Apple была и остается лидером в области графического интерфейса пользователя. Сама концепция WIMP-интерфейса, если не родилась в компьютерах Macintosh, то прошла на них промышленную апробацию. Mac OS является системой с "только-графическим" интерфейсом, и все операции пользователя с системой и с приложениями

производятся только через манипулирование графическими объектами.

Основные концепции интерфейса Mac OS:

- метафоры, главной из которых является метафора рабочего стола и интегрированные с ней метафоры документов и папок;
- прямое манипулирование объектами;
- прямое целеуказание (принцип see-and-point – увидеть и указать);
- согласованность интерфейса – одинаковое образное представление и одинаковое поведение интерфейсных элементов в разных приложениях и системных операциях;
- доступность всех объектов и функций для пользователя;
- информированность пользователя о происходящих в системе процессах и о результатах его воздействий;
- и т.д.

Эти и другие свойства обеспечивают интерфейсу Mac OS интуитивную понятность, дружелюбность и предсказуемость. Большинство принципов построения пользовательского интерфейса Mac OS было с той или иной степенью последовательности внедрено в другие системы, так что даже пользователи, никогда не имевшие дела с компьютерами Macintosh, имеют о них представление "из вторых рук". Следует, однако, отдельно упомянуть еще об одном из таких принципов, отличающем интерфейс Mac OS от интерфейса, например, Windows. Это принцип "пользовательского управления". Apple исходит из того, что работа происходит более эффективно, если пользователь является в ней активной, инициативной стороной. Это означает, что пользователь, а не компьютер должен инициировать управляющие действия. Разумеется, в некоторых случаях компьютер "берет управление на себя" – если, например, имеются ограничения в выборе альтернатив в данном контексте или для того, чтобы предохранить пользователя от излишней детализации

в формировании управляющего действия. Однако подход Apple состоит в соблюдении такого баланса между предоставлением пользователю всей полноты возможностей и предохранении его от разрушения данных, при котором инициатива пользователя не ущемляется никоим образом.

Набор программных модулей, которые обеспечивают базовые функции работы с интерфейсной графикой (рисование, перемещение, вращение и т.п. образов), называется в Mac OS QuickDraw. Приложения используют QuickDraw неявным образом, когда обращаются к другим менеджерам графического интерфейса для создания интерфейсных элементов и манипулирования ими. QuickDraw развивался вместе с Mac OS – от черно-белого изображения к цветному и далее – к 3-мерной графике. При этом сохраняется совместимость новых версий QuickDraw со всеми предыдущими. Современные версии QuickDraw GX используют объектно-ориентированную архитектуру графического интерфейса.

Хотя Mac OS является однопользовательской системой с невывесняющей многозадачностью, она вполне удовлетворяла требованиям своих пользователей, несмотря на то, что некоторые применяемые в ней технологии уже несколько устарели. Mac OS версии 8 и 9 и сейчас продолжает применяться. Однако расширение ресурсов компьютеров Macintosh и стремление фирмы Apple выйти на рынок серверных систем потребовали создания принципиально иной ОС.

8.3. Mac OS X

Путь фирмы Apple к новой операционной системе – многопользовательской, с вытесняющей многозадачностью и с защитой памяти – оказался долгим и нелегким. В середине 90-х годов фирма неоднократно начинала проекты новых ОС, иногда даже демонстрировала их альфа-версии, но по разным причинам эти проекты так и не дошли до

промышленной реализации. Только в 1998 г. усилия фирмы увенчались успехом и появилась новая ОС – Mac OS X [29]. Текущая версия Mac OS X – 10.1.3 (сохранена сквозная нумерация версий от Mac OS к Mac OS X).

Mac OS X представляет собой удивительное сочетание оригинальных закрытых программных технологий Apple с открытыми технологиями, ставшими промышленными стандартами. Архитектура Mac OS X, показанная на рисунке 8.3, имеет явно выраженную иерархическую структуру.

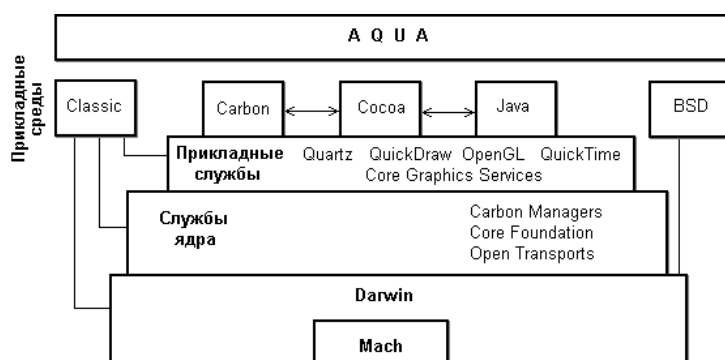


Рисунок 8.3 Архитектура Mac OS X

Микроядро Darwin

Mac OS X строится на базе микроядра, которое называется Darwin. Внутри же Darwin находится "ядро в ядре" – микроядро Mach. Mach [27] является "классическим" микроядром, оно было разработано в университете Carnegie–Mellon (начало проекта – 1985 г.), и именно в этом проекте родились основные концепции архитектуры микроядра, ныне являющиеся общепринятыми. Микроядро Mach было создано на основе BSD и послужило основой для ряда Unix-подобных (точнее – BSD Unix-подобных) систем, например, ядра OSF/1 и сделанной на его основе ОС DIGITAL UNIX.

И Mach, и Darwin являются продуктами в Открытых Кодах и поддерживаются организацией Open Group.

Mac OS X строится на версии микроядра Mach 3 и, по-видимому, является единственной не-Unix системой, использующей ядро Mach.

Mach поддерживает основные низкоуровневые функции управления ресурсами, такие как:

- управление единицами выполнения (нитеями);
- назначение ресурсов для процессов (в терминологии Mach – задач, task);
- поддержку адресных пространств для задач;
- обмен сообщениями между задачами;
- управление реальными ресурсами (процессорами, памятью, вводом-выводом).

Управление памятью в Mach, как и в большинстве современных Unix-систем, обеспечивает для каждой задачи виртуальное адресное пространство размером 4 Гбайт, в принципе, изолированное от адресных пространств других задач. Адресное пространство строится на страничной модели памяти, однако соседние виртуальные страницы, обладающие одинаковыми свойствами, могут составлять область (сегмент). Как и многие другие Unix-системы, Mach использует абстракцию "объектов памяти", представляющую собой надстройку над обычными механизмами виртуальной памяти. Объекты памяти создаются в виртуальном адресном пространстве, а реальная память рассматривается только как кеш для представления этих объектов. Области и объекты памяти могут совместно использоваться несколькими задачами.

Mach обеспечивает вытесняющую многозадачность и многопоточность (API нитей в Mach соответствует спецификациям POSIX). Как и во всех BSD-системах нить обладает достаточно полным набором ресурсов для выполнения, таким образом, в Mach нет необходимости вводить легковесные процессы, как, например, в Open

Unix. Все нити одной задачи разделяют адресное пространство задачи и некоторые ресурсы задачи. Каждая нить имеет собственный вектор состояния, стек, параметры планирования и коммуникационные порты. Диспетчеризация нитей ведется по приоритетному принципу, приоритеты назначаются и изменяются вне микроядра. Нить может быть сделана "закрепленной" (wired). Такая нить является привилегированной: она получает управление сразу же при достижении состояния готовности и ей выделяется память даже при нехватке реальной памяти. Это позволяет Mach обеспечивать процессы реального времени.

Многопоточность Mach работает как на одном процессоре, так и на SMP конфигурациях.

Задачи в Mach взаимодействуют через посылку сообщений и прием ответов. Сообщения передаются через коммуникационные порты, которые представляют собой почтовые ящики или очереди сообщений, описанные нами в главе 9 части I. При создании любой нити для нее создается также собственный порт для приема сообщений от других нитей и порт для приема исключений. Собственный набор портов создается и для задачи.

Микроядро Darwin является расширением Mach. Кроме Mach Darwin содержит следующие основные компоненты:

- Инструменты ввода-вывода – объектно-ориентированный каркас для разработки драйверов устройств, создания драйверов и обеспечения требуемой для драйверов инфраструктуры.
- Файловая система – основывается на виртуальной файловой системе VFS и обеспечивает возможность добавлять новые файловые системы. В настоящее время поддерживаются HFS, HFS Plus, ufs и ISO 9660 – файловая система для CD.
- Расширенные сетевые средства Network Kernel Extensions (NKE), позволяющие разработчикам как добавлять поддержку новых

протоколов, так и расширять функциональность уже поддерживаемых.

- BSD – оболочка BSD 4.4 вокруг ядра. Реализация BSD в Darwin включает в себя много API POSIX, обеспечивает модель процессов, базовые политики безопасности и поддержку нитей для Mac OS X.

Службы ядра

Службы ядра содержат те системные сервисы, которые не связаны с графическим интерфейсом пользователя. Основные компоненты этих служб – менеджеры среды Carbon, а также Core Foundation и Open Transport.

Менеджеры среды Carbon являются общесистемными и обеспечивают низкоуровневый сервис для всех прикладных сред. В число этих менеджеров входят, например:

- Collection Manager – обеспечение абстрактных типов для коллекций данных.
- Component Manager – обеспечение для приложения возможности находить во время выполнения различные программные объекты (компоненты), а также создавать компоненты.
- Date, Time, and Measurement Utilities – работа с датой, временем, географическими местами, временными зонами и т.п.
- File Manager – файловый API для всех файловых систем.
- Folder Manager – обеспечение работы с папками.
- Memory Manager – выделение памяти в виртуальном адресном пространстве задачи и другие функции управления виртуальной памятью.

- Multiprocessing Services – средства для создания нитей, управления ими и синхронизации.

Core Foundation – каркас, который обеспечивает некоторые базовые программные службы, полезные для более высоких уровней программного обеспечения. Core Foundation использует объектно-ориентированную парадигму "непрозрачных" типов, "черных ящиков" для таких программных объектов как числа, строки, массивы, словари, деревья и т.д. Этот компонент также обеспечивает работу с подключениями (plug-in) и ряд других сервисов. Некоторые из сервисов, обеспечиваемых Core Foundation:

- String Services – набор инструментов для манипулирования строками, включая поддержку
- Unicode.
- Bundle Services – средства организации и поиска различных типов программных ресурсов (исполняемых кодов, графических и звуковых образов и т.п.).
- Plug-in Services – обеспечение архитектуры подключений.
- Collection Services – высокоуровневые абстракции коллекций.
- URL Services – средства доступа к локальным или удаленным ресурсам через URL.
- Notification Services – механизм обмена сообщениями (уведомлениями) между процессами.

Open Transport – основные модули пользовательского уровня для обеспечения работы в сети и коммуникаций в Mac OS X.

Прикладные службы

Главная задача прикладных служб Mac OS X – обеспечение графического и оконного интерфейса. Главной частью этих служб является

набор модулей Quartz, который состоит из двух частей: исполнения изображений (показано на рисунке 8.3 как собственно Quartz) и базовых графических служб или сервера окон (показано на рисунке 8.3 как Core Graphics Services). Вторая часть представляет собой библиотеку, обеспечивающую некоторые общие сервисы для других прикладных служб.

В сумме Quartz является мощной графической системой, которая обеспечивает 2-мерную графику на основе формата PDF и работу с окнами и составляет основу для формирования изображений в Mac OS X – как для системных модулей, так и для приложений. Эта система обладает такими свойствами, как независимость от разрешающей способности, преобразование координат, сплайны, прозрачность, сглаживание и т.д., что позволяет обеспечить системе и приложениям весьма изысканный графический интерфейс.

Mac OS X реализует также OpenGL – многоплатформенный промышленный стандарт для 3-мерного рисования и ускорения работы аппаратуры. Использование OpenGL обеспечивает высокую эффективность в создании анимации в реальном времени для игр и научной или деловой визуализации.

Система QuickTime предоставляет средства для эффективной работы с мультимедийной информацией, такой как видеоролики, изображения, аудиозаписи. Компоненты QuickTime позволяют приложениям не зависеть в работе с мультимедийной информацией от конкретных типов устройств и памяти. Каждый компонент обеспечивает какой-то определенный набор свойств и предоставляет определенный API для использующих его приложений. Компонент является кодовым ресурсом, который регистрируется Менеджером Компонентов, и Менеджер Компонентов обеспечивает его доступность в рамках всей системы. Различные

приложения могут использовать компоненты, не вникая в детали их реализации.

Некоторые прикладные службы Mac OS X (не показанные на рисунке 8.3) обеспечивают отображение низкоуровневых объектов (объектов ядра) в объекты API. Менеджеры Carbon, которые обеспечивают этот сервис, обслуживают все прикладные системы. Ниже мы рассматриваем некоторые из этих служб.

Carbon Process Manager (CPM) обеспечивает абстракцию процесса для прикладных сред. В ядре процесс (задача) является сущностью, состоящей из набора нитей, адресного пространства и пространства имен портов. CPM на базе задачи ядра создает CPM-процессы, которые представляют процессы для прикладных сред. В средах Carbon, Cocoa и Java каждому CPM-процессу соответствует одна задача ядра. Для среды Classic (с невытесняющей многозадачностью) создается один CPM-процесс для каждого приложения, но все CPM-процессы приложений отображаются на единственную задачу ядра.

Базовый механизм нитей в микроядре Mach преобразуется в ядре в многопоточную среду POSIX. нитям микроядра соответствуют нити POSIX. Лежащие выше уровни программного обеспечения создают прикладные модели многопоточных сред, а именно:

- Multiprocessing Service – диспетчеризация нитей с вытеснением в среде Carbon;
- Tread Manager – диспетчеризация нитей без вытеснения в среде Carbon;
- NSThread – класс-оболочка для представления нитей с вытеснением в среде Cocoa;
- java.lang.Thread – класс-оболочка для представления нитей с вытеснением в среде Java.

Во всех моделях кроме Tread Manager нити приложения соответствует нить POSIX, в модели Tread Manager все нити приложения отображаются на одну нить POSIX.

Базовые механизмы в ядре, обеспечивающие взаимодействие между процессами, – очереди сообщений, передаваемых через коммуникационные порты. Прикладные службы строят на основе этого механизма множественные прикладные модели взаимодействия, а именно:

- события Apple;
- простые уведомления (simple notification) – передача сообщения в "центр уведомлений", который распространяет сообщение для всех процессов, которые в нем "заинтересованы";
- передача неструктурированных данных – быстрый низкоуровневый способ обмена данными между локальными процессами;
- сокеты BSD – основной механизм Mac OS X для передачи данных в сети;
- программные каналы (pipe);
- сигналы (набор сигналов BSD);
- разделяемые области памяти с управлением доступом к ним через семафоры;
- объектно-ориентированные механизмы "стандартных служб" и "распределенных объектов" в среде Cocoa;
- обмен сообщениями через порты микроядра Mach.

Прикладные среды

Прикладные среды Mac OS X состоят из каркасов (framework), библиотек и сервисов, которые обеспечивают выполнение приложений в

той или иной модели API. Mac OS X в настоящее время поддерживает следующие прикладные среды.

- Carbon – развитие API Mac OS для Mac OS X. Около 70% системных вызовов Carbon имеются и в Mac OS, таким образом, может быть обеспечена переносимость приложений в обе стороны. Как было показано выше, Менеджеры Carbon выполняют обслуживание также и других прикладных сред. В Carbon часть менеджеров Mac OS подверглась усовершенствованию, часть была заменена, некоторые были добавлены. Наиболее существенные изменения произошли в управлении памятью (адаптация к более развитой модели виртуальной памяти и к защите памяти), в интерфейсах оборудования (менеджеры Mac OS X уже не выполняют низкоуровневые операции на оборудовании непосредственно), полностью заменены менеджеры печати и управления событиями.
- Cocoa – объектно-ориентированная среда для языков Java и Objective-C. Базируется на двух каркасах: Foundation и Application Kit. Каркас Foundation обеспечивает объекты и методы, не связанные напрямую с интерфейсом: базовые типы и операции (строки, массивы, словари и т.п.), классы-оболочки для объектов ядра (задачи, нити, порты и т.д.), общую функциональность, связанную с объектами (управление памятью, архивация, сериализация и т.д.), функциональность ввода-вывода и файловой системы, другие службы (распределенные уведомления, дата и время, откат операций и т.д.). Каркас Application Kit в основном обеспечивает классы пользовательского интерфейса (окна, меню, диаголи, кнопки и т.п.), но также и набор более развитых возможностей, таких как рисование и создание композитных

образов, управление событиями, приложениями и документами и т.д.

- Java позволяет разрабатывать и выполнять в Mac OS X приложения и апплеты, соответствующие спецификациям 100% "чистой" Java. Среда Java включает в себя компилятор `javac` и полный набор утилит, среду выполнения – виртуальную машину Java, базовый набор пакетов Java и компилятор байт-кода в коды целевой платформы, пакеты `awt` и `swing`.
- Среда Classic обеспечивает выполнение приложений Mac OS. В этой среде не обеспечиваются свойства, предоставляемые новым ядром ОС и интерфейсом Aqua. Эта среда не поддерживается прикладными службами непосредственно, следовательно, она обеспечивает только выполнение приложений, но не их разработку.
- Среда BSD выполняет программы BSD из командной строки. Она обеспечивает `shell` и стандартный набор команд и утилит BSD. Эта среда базируется непосредственно на функциях ядра и не является обязательной для Mac OS X (может быть отменена при инсталляции).

Интерфейс Aqua

Интерфейс Mac OS X называется Aqua. В этой разработке фирма Apple вновь доказала, что она является лидером в продвижении дружественных графических интерфейсов. Aqua с одной стороны наследует интерфейсу Mac OS, а с другой – предлагает новые функциональные возможности и новый дизайн.

Среди новых функциональных возможностей Aqua можно назвать такие, как:

- введение нового объекта, который называется Док (Dock – бассейн для стоянки кораблей), для отображения иконок открытых приложений и минимизированных документов, его функциональность отчасти та же, что и у Линейки Программ, но навигация в Доке гораздо более удобна;
- введение полотнищ (sheet) – диалогов, привязанных к своему окну, что позволяет сделать диалоги немодальными;
- введение иерархии окон, облегчающей ориентацию в монгооконной среде;
- возможность масштабирования иконок от максимального размера 128x128 до мини-иконок;
- введение "выдвижных ящиков" (drawer), содержащих те управляющие элементы окна, в постоянной визуализации которых нет необходимости;
- использование анимации для отображения изменения состояния элементов интерфейса;
- расширение возможностей использования клавиатуры;
- и т.д., и т.п.

Дизайн же нового интерфейса Mac OS X представляется почти революционным. Название нового интерфейса отражает метафору, послужившую основой для внешнего вида интерфейса. В изображении повсеместно используется метафора воды с такими замечательными свойствами этой субстанции, как цвет, глубина, прозрачность, блики, движение. Кнопки выглядят как отполированные и блестящие цветные стеклышки, все объекты отбрасывают размытые тени, меню просвечиваются, диалоги полупрозрачны и т.д. Дизайнеры интерфейса смогли почувствовать и воплотить на экране ту одухотворенность воды, которая на протяжении многих веков вдохновляла художников и поэтов.

При этом, как это свойственно интерфейсам Apple, изобразительные средства ни в коей мере не мешают функциональности, а наоборот – подчеркивают и усиливают ее.

В настоящее время фирма предлагает компьютеры iMac, iBook, PowerBook, Server G4 на процессорах PowerPC поколений G3 и G4 – от ноутбуков до профессиональных графических станций и серверов. Все компьютеры Apple работают под управлением ОС Mac OS X. Хотя фирма Apple далека от каких-либо претензий на господствующее положение на рынке, ее аппаратная и программная продукция – факт, с которым приходится считаться всем производителям информационных технологий, претендующих на обеспечение совместимости.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Почему, по-вашему, Mac OS, не обеспечивающая вытесняющую многозадачность, просуществовала так долго?
2. Укажите общие черты управления памятью в PalmOS и в 24-разрядной модели Mac OS.
3. Как Mac OS борется с внешними дырами в памяти?
4. Какие из описанных в Части I механизмов функционирования ОС положены в основу модели выполнения CFM Mac OS?
5. В чем отличия модели выполнения CFM от классической модели M68K?
6. Что в Mac OS называется дополнением (plug-in)?
7. Как приложение Mac OS может обрабатывать прерывания? Сопоставьте в этом отношении Mac OS и MS DOS.
8. Что представляет собой "толстый" ресурс в Mac OS?
9. Опишите механизмы разных уровней, обеспечивающие взаимодействие процессов в Mac OS.

10. Какой ветви семейства ОС Unix в наибольшей степени соответствует микроядро Mach?

11. Опишите структуру Mac OS X и охарактеризуйте ее компоненты. Как можно классифицировать архитектуру Mac OS X?

12. Какие средства взаимодействия процессов унаследованы Mac OS X от Mac OS? Какие новые средства введены?

13. Какие прикладные среды поддерживает Mac OS X?

14. Как обеспечивается работа приложений, созданных для Mac OS в среде Mac OS X?

Глава 9. Операционная система BeOS

9.1. Короткая история и позиционирование системы

Операционная система BeOS [36] разработана фирмой Be Inc., созданной в 1990 г. Первоначально фирма производила также и собственные компьютеры BeBox на базе процессора AT&T Hobbit, однако компьютеры Be не утвердились на рынке и сейчас компания специализируется на программном обеспечении – BeOS и созданном на ее основе пакете BeIA – интегрированном средстве работы в Internet.

Программное обеспечение Be работает на платформах Intel-Pentium, PowerMac и PowerPC. Последним релизом BeOS является версия 5. BeOS v.5 для некоммерческого использования распространяется свободно.

В основу создания BeOS была положена концепция "молодой" ОС, которая не будет обременена многолетним наследством и с самого начала будет построена с учетом некоторых реалий современных подходов к обработке информации – прежде всего мультимедийной информации и Internet. Наверное, в начале истории BeOS ее создатели имели "тайную

мысль" создать универсальную ОС для настольного применения. Но выходить на рынок с таким предложением было рискованно, поэтому для новой ОС была найдена "экологическая ниша", в которой, по крайней мере в то время, у BeOS опасных конкурентов почти не было. Этой нишей стала разработка и выполнение мультимедийных приложений. Ориентация на мультимедиа была заложена в самые основы BeOS и продолжала развиваться во всех последующих версиях. В ходе дальнейшего развития BeOS так и не удалось выйти из своей первоначальной экологической ниши. Основной причиной этого, как обычно, называют отсутствие на платформе BeOS достаточного числа известных пользователям приложений. Более того, все более широкое обеспечение универсальных ОС (прежде всего – Windows) мультимедийными приложениями создает серьезную конкуренцию для BeOS и в ее собственной нише. В настоящее время фирма Be пытается внедриться в рынок Internet-вычислений, и эта ее попытка вполне оправдана, так как роль мультимедийной информации в этой области весьма велика и продолжает расти.

Как раз в те дни, когда писалась эта глава, на сайте компании Be появилось сообщение о том, что права интеллектуальной собственности на технологии Be куплены фирмой Palm Inc. Предполагается, что за этим последует полная интеграция Be Inc. в Palm.

Ориентация на мультимедиа наложила определенный отпечаток на структуру BeOS. Подобно QNX, BeOS строится на базе микроядра и процессов-серверов. Независимые серверы в сочетании с объектно-ориентированной структурой системы обеспечивают для ОС гибкость и простоту в наращивании функциональности.

Поскольку задачи обработки мультимедийной информации эффективно решаются методами распараллеливания, в BeOS уделяется большое внимание эффективному использованию многопроцессорных конфигураций. Основной концепцией BeOS является "всепроникающая

многопоточность" – возможность для приложений создавать практически неограниченное число нитей и интенсивное использование распараллеливания на уровне нитей во всех сервисах самой ОС – в графической системе, вводе-выводе, файловой системе. При работе на платформе PowerPC BeOS в полной мере использует обеспечиваемое аппаратурой процессора 64-разрядное адресное пространство, что также существенно для мультимедиа.

BeOS обеспечивает API POSIX, однако нас интересуют прежде всего ее оригинальные системные интерфейсы. К сожалению, сколько-нибудь доступная информация о внутренней структуре BeOS не публикуется. Приводимые далее материалы в основном почерпнуты нами из информации для разработчиков приложений. Однако и они позволяют делать некоторые выводы (пусть косвенные) об устройстве BeOS. Следует отметить, что по своей структуре BeOS является объектно-ориентированной системой, поэтому в ней существует "двойная бухгалтерия" системных вызовов: они могут выполняться через обращения к библиотечным функциям C – и так реализованы интерфейсы POSIX, но могут выполняться также и через обращения к методам библиотечных объектов C++. Оба способа обеспечивают одинаковую функциональность практически во всем.

9.2. Поток и команды

BeOS является многопоточной системой с несколько оригинальной концепцией распределения и разделения ресурсов. Ключевым понятием BeOS является нить. С точки зрения распределения процессорного времени нить BeOS идентична нити в других системах: нить является субъектом, для которого планируется процессорное время. Однако, понятия процесса в BeOS нет. Наиболее близким к нему является понятие

команды (team). Команда представляет собой группу нитей, составляющих одно приложение. При запуске приложения на выполнение (оператором или другим приложением) для него создается нить, составляющая новую команду, и в этой нити выполняется функция `main()`. Нить `main` может порождать другие нити. Все нити разделяют общее адресное пространство и используют общие глобальные для приложения переменные.

Новая нить порождается системным вызовом `spawn_thread()`, которому передается имя той функции, которая будет выполняться в нити и указатель на блок параметров функции нити. Созданная таким образом нить еще не выполняется. Она может быть запущена на выполнение системным вызовом `resume_thread()` или `wait_for_thread()`. В первом случае нить-потомок выполняется асинхронно, то есть параллельно с нитью, ее породившей. Второй случай – синхронный запуск, выполнение породившей нити приостанавливается до завершения нити-потомка.

Мы говорим о нитях – родителе и потомке, однако на самом деле родственные отношения между порождающей и порожденной нитью весьма слабы. Системный вызов `spawn_thread()` возвращает нити-родителю идентификатор нити-потомка, но не обеспечивает наследования никаких ресурсов, кроме общих для всей команды. Нити выполняются независимо друг от друга, и выполнение нити-потомка продолжается даже после завершения родительской нити. Теоретически даже завершение нити, в которой выполняется функция `main()` не приводит к завершению всех порожденных ею нитей. Но именно нити `main` выделяют общие для всей команды статические объекты и ресурсы ввода-вывода, так что завершение нити `main` скорее всего приведет к аварийному завершению остальных нитей команды.

При создании нити ей может быть дано имя. Другая нить, желающая обратиться к данной, независимо от того, находится она в этой же команде

или в другой, может использовать системный вызов `find_thread()`, который по имени нити возвращает ее идентификатор. Но идентификатор нити является уникальным во всей системе, а имя нити – не уникально. Вызов `find_thread()` возвращает идентификатор первой найденной нити с таким именем. Поэтому более надежным способом получения идентификатора нити является передача его нити-корреспонденту через глобальные переменные, параметры, средства взаимодействия и т.п.

Все нити выполняются параллельно, разделяя процессор (или процессоры) в соответствии с приоритетами. Приоритеты со значениями от 1 до 99 составляют класс приоритетов деления времени, приоритеты со значениями 100 и выше – класс приоритетов реального времени.

Приоритеты деления времени относительные – нити с такими приоритетами выполняются в режиме квантования времени с размером кванта 3 мсек. Приоритет определяет частоту получения кванта нитью. Нить, получившая квант, использует процессор до исчерпания ею кванта или до перехода в ожидание по собственной инициативе, или до появления нити с приоритетом реального времени. Нити деления времени не вытесняют друг друга.

Приоритеты реального времени абсолютные. Когда нить с приоритетом реального времени приходит в состояние готовности, она немедленно вытесняет с процессора нить с приоритетом деления времени или нить с более низким приоритетом реального времени.

Приоритеты являются статическими: они задаются при создании нити и не изменяются в дальнейшем.

Нить может до некоторой степени управлять своим планированием, переходя в состояние приостанова на заданный интервал времени (системный вызов `snooze()`) или завершаясь (системный вызов `exit_thread()`).

Управление нитью "со стороны" – из другой нити, которой известен идентификатор управляемой, возможно следующее:

- нить может быть приостановлена системным вызовом `suspend_thread()`, а затем вновь запущена на выполнение системным вызовом `resume_thread()` или `wait_for_thread()`.
- для запуска заблокированной или "спящей" нити может быть использован системный вызов POSIX `send_signal()`. Сигнал `SIGCONT` разблокирует нить.
- системный вызов `kill_thread()` прекращает выполнение нити.

9.3. Средства взаимодействия

При создании каждой нити для нее создается буфер сообщения. Другая нить, зная идентификатор нити-корреспондента может записать в этот буфер сообщение системным вызовом `send_data()`, а нить – владелец буфера выбирает сообщение системным вызовом `recv_data()`. Однако буфер рассчитан только на одно сообщение, а попытки писать данные в занятый буфер или выбирать данные из пустого буфера приводят к блокировке нити.

Более гибким средством обмена данными между нитями является порт (port). Следует отметить, что порт не является прямым аналогом ни одного из средств взаимодействия процессов, рассмотренных нами в главе 9 части I. Порт представляет собой общесистемную очередь сообщений, работающую по дисциплине "первым пришел – первым вышел". В системе может быть создано сколько угодно портов. Любая нить из любой команды, которой известен идентификатор порта, может записать в порт

сообщение (системный вызов `write_port()`) и прочитать из порта сообщение (системный вызов `read_port()`). При создании порта порт сообщение (системный вызов `create_port()`) задается его емкость – число сообщений, которое может храниться в порте. Попытка писать в переполненный порт или читать из пустого порта, естественно, приводит к блокировке нити. Однако, есть варианты системных вызовов `write_port_etc()` и `read_port_etc()`, которые к блокировке не приводят. Однако система поддерживает общий репозиторий портов, емкость которого равна суммарной емкости всех созданных портов, и переполнение происходит только при заполнении общей емкости.

Порт принадлежит команде, в которой он был создан. Однако, если идентификатор порта, возвращаемый системным вызовом `create_port()`, передается в другую команду, эта другая команда также может использовать порт. Системный вызов `delete_port()` уничтожает порт, системный вызов `close_port()` закрывает порт для записи, но оставляет возможность прочитать сообщения, еще остающиеся в порте. Порт автоматически уничтожается, когда завершается последняя нить команды, в которой он был создан. Однако создавшая порт команда может передать право владения портом другой команде системным вызовом `set_port_owner()`.

Еще раз подчеркнем, что порт является только FIFO-очередью и никаких средств неразрушающего чтения из порта не существует.

Семафоры в BeOS представляют собой традиционные общие семафоры. Семафор создается системным вызовом `create_sem()`, системные вызовы `acquire_sem()` и `release_sem()` обеспечивают традиционные семафорные операции P и V соответственно. Начальное значение семафора задается при его создании, но значение семафора может и превысить начальное, если операции `release_sem()`

выполняются чаще, чем `acquire_sem()`. Семафор принадлежит той команде, в которой он был создан, и автоматически уничтожается с завершением последней нити этой команды. Явным образом семафор может быть уничтожен системным вызовом `delete_sem()`. Идентификатор семафора, который был возвращен вызовом `create_sem()`, может быть передан в другую команду, но право владения семафором не передается.

9.4. Управление памятью

В части управления памятью BeOS обеспечивает сегментную модель для приложений, однако, в ней "просматривается" сегментно-страничная реализация. Любая нить может запросить выделение для нее области (area) памяти. Область представляет собой непрерывный участок виртуальной памяти, размер которого задается в системном вызове `create_area()`. Размер области должен быть кратен размеру страницы (4 Кбайт). Операция создания области возвращает ее адрес в виртуальном адресном пространстве команды. При создании области нить может явно задать адрес в своем виртуальном адресном пространстве, по которому область должна быть размещена, но адрес размещения области обязательно выравнивается по границы страницы. Кроме того, при создании каждой новой области ей присваивается уникальный во всей системе идентификатор. Этот идентификатор может использоваться в вызове `delete_area()` или передаваться другим командам для совместного использования области.

Области также может быть присвоено имя. Системный вызов `find_area()` возвращает идентификатор области с заданным именем. Однако, как и в случае с нитями, имя области не является уникальным, и

системный вызов `find_area()` возвращает идентификатор только первой найденной области.

В пределах одной команды все нити "видят" созданную область по одному и тому же виртуальному адресу. При совместном использовании области двумя и более командами команда, не являющаяся владельцем области, должна получить ее идентификатор и "клонировать" область при помощи системного вызова `clone_area()`. Параметром этого вызова является идентификатор области, а возвращает он виртуальный адрес области. Этот адрес может отличаться от виртуального адреса области в той команде, в которой область была создана. Клонирование области, однако, не означает дублирования ее данных. Оно просто задает отображение участков виртуального адресного пространства разных команд на одну и ту же реальную память. Изменения в содержимом области, сделанные одной командой, будут немедленно видны в другой команде.

Область явно уничтожается системным вызовом `delete_area()` или неявно – при завершении всех нитей команды, в которой область была создана. Если, однако, область была клонирована, то ее реальное освобождение происходит при уничтожении (явном или неявном) ее последнего клона.

При создании или клонировании области можно сделать ее защищенной от записи или защищенной от чтения.

При создании области можно также зафиксировать ее в реальной памяти, при этом имеются возможности:

- выделить для области физическую память немедленно и исключить ее из страничного обмена;

- выделять для области физическую память постранично, по требованию и выделенные страницы исключать ее из страничного обмена;
- выделить для области физическую память немедленно, причем в непрерывной области реальной памяти, и исключить ее из страничного обмена.

9.5. Образы

Программные коды, готовые для выполнения, называются в BeOS образами (image). Различаются три вида образов:

- образы приложений;
- библиотечные образы;
- дополнительные (add-on) образы.

Образы приложений являются загрузочными модулями программ. Для их загрузки и связывания применяется системный вызов `load_image()`. Параметром вызова является имя файла, из которого загружается образ приложения. Этот вызов в чем-то подобен вызову `spawn_thread()`. Он также создает новую нить. В этой нити будет выполняться функция `main()` приложения. Но этот вызов создает также и новую команду, "возглавляемую" нитью `main` запускаемого приложения, а следовательно, и новое виртуальное адресное пространство и другие общекомандные ресурсы. Как и `spawn_thread()`, `load_image()` возвращает идентификатор нити. Созданная таким образом нить должна быть запущена на выполнение теми же системными вызовами `resume_thread()` или `wait_for_thread()`.

Библиотечные образы являются модулями динамической компоновки, подключение (связывание) которых происходит

автоматически при загрузке приложения. Библиотечные образы используются совместно всеми приложениями.

Дополнительные образы также являются совместно используемыми модулями динамической компоновки. Но их загрузка и связывание происходят по требованию, уже в процессе выполнения приложения. Загрузка такого модуля выполняется при помощи системного вызова `load_add_on()`, которому передается имя файла, содержащего дополнительный образ. Вызов `load_add_on()` возвращает идентификатор загруженного образа, который далее можно использовать в качестве параметра системного вызова `get_image_symbol()`, чтобы получить адреса внешних символов и входных точек дополнительного образа.

С точки зрения формата дополнительные образы ничем не отличаются от библиотечных. В параметрах системного окружения отдельно задаются каталоги, из которых загружаются библиотечные и дополнительные образы. Манипулируя этими параметрами, можно сделать так, что образ, к одному приложению подключаемый при загрузке, для другого будет дополнительным.

9.6. Устройства и файловые системы

Драйверы в BeOS являются расширениями ядра системы и могут работать в адресном пространстве ядра. В системе различаются три вида драйверов:

- драйверы устройств;
- драйверы файловых систем;
- модули.

Последние являются вспомогательными программными единицами, выполняющими некоторый общий сервис, необходимый для всех или нескольких драйверов устройств (например, управление шиной SCSI). Если первые два вида драйверов доступны для пользовательских приложений через API, то модули полностью скрыты от пользователей и вызываются только из других драйверов.

Обращения к драйверам из приложений выполняются через API POSIX (`open()`, `read()`, `write()` и т.д.). Перевод API POSIX во внутренние системные вызовы ядра BeOS осуществляет "файловая система устройств" `devfs`. Для того, чтобы драйвер был доступен для `devfs`, он должен быть записан (опубликован) в соответствующем каталоге иерархической файловой системы.

К сожалению, нам не удалось найти исчерпывающей информации о файловой системе BeOS, но даже та неполная информация, которую нам удалось получить, представляет существенный интерес.

Первая файловая система BeOS не имела иерархической структуры. Вместо этого логическая структура файловой системы поддерживалась "базой данных файлов". Навигация по логической структуре осуществляется средствами, напоминающими средства, принятые в реляционных базах данных – запросами. Поиск файла выполняется запросом, содержащим предикат (иногда довольно сложный), проверяющий значение одного или нескольких атрибутов файла. Хотя бы для одного из атрибутов, участвующих в предикате, должен быть создан индекс. Наряду с запросами, формулируемыми при каждом обращении, в системе существуют и "постоянно живущие" запросы – аналоги представлений (`view`) в реляционных базах данных. Подобно представлению, постоянный запрос представляет собой не зафиксированную выборку, а зафиксированный предикат, выборка же при каждом обращении выполняется заново. Постоянные запросы

представляют собой функциональный аналог каталогов в иерархической файловой системе.

В 1997 г. для BeOS была разработана новая файловая система – BFS. В ней была введена иерархическая структура файловой системы и логическая структура в значительной степени интегрировалась с физической структурой хранения. Однако наряду с иерархической логической структурой BFS поддерживает и индексирование по именам и другим атрибутам файлов и, таким образом, в полном объеме сохраняется возможность альтернативной "реляционной" навигации по файловой системе.

Единицей распределения дискового пространства является блок, размер блока выбирается из ряда: 512 байт, 1 Кбайт, 2 Кбайт, 4 Кбайт, 8 Кбайт. Файл состоит из одного или нескольких экстенгов, каждый экстенг – целое число последовательных блоков. План размещения файла представляет собой массив описателей экстенгов. Каталоги структурированы в виде B^+ -деревьев. Дескрипторы файлов и элементы каталогов разделены, но несмотря на это, BFS не поддерживает "жесткие" связи (алиасы), а только "мягкие" связи (косвенные файлы). В дескрипторе файла хранятся основные его атрибуты, расширенные же атрибуты хранятся вместе с данными файла.

С введением BFS была введена и концепция виртуальной файловой системы, обеспечивающая для BeOS возможность работы с файловыми системами различных форматов (CDFS и HFS от MacOS). Ядром BeOS формируется корневой каталог виртуальной файловой системы, в котором не могут находиться файлы, а только подкаталоги и "мягкие" связи. Другие физические файловые системы монтируются как подкаталоги корневого каталога. Ряд подкаталогов и связей монтируются в корневой каталог автоматически, при загрузке системы. Также автоматически монтируются и еще две виртуальные файловые системы: /dev – виртуальная файловая

система устройств и /pipe – виртуальная файловая система программных каналов.

Некоторые другие интересные свойства BFS также определяются спецификой этой файловой системы:

- 64-разрядный размер файла – важное обстоятельство, если учесть то, что многие файлы в BFS содержат мультимедийную информацию;
- использование многопоточности в работе самих модулей BFS – в соответствии с общей концепцией всей BeOS;
- журналирование – свойство, которое может показаться роскошью для настольной ОС, но в BFS оно совершенно необходимо для сохранения целостности при сбоях базы данных файлов.

Интересен способ, который использует BeOS для определения типа файла, а следовательно, и приложения, по умолчанию связываемого с визуализацией и обработкой этого файла. В атрибутах файла обеспечивается идентификация типа в соответствии со спецификациями MIME (Multipurpose Internet Mail Extensions). Наряду со стандартными типами MIME BeOS применяет также и собственные типы, не предусмотренные в MIME, но определяемые также в формате спецификации MIME. При отсутствии у файла MIME-специфицированных атрибутов для определения типа используется расширение файла, и BeOS ведет собственную "базу типов файлов", которые определяют связанные с типом-расширением приложения. BeOS также представляет пользователю возможность назначать собственные интерпретации типа для каждого файла или группы файлов.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Переведите терминологию BeOS, относящуюся к управлению процессами и нитями в принятую в части I данного учебного пособия.
2. Какие оригинальные средства взаимодействия процессов применяются в BeOS?
3. Какую модель памяти обеспечивает API BeOS?
4. Сопоставьте "образы" BeOS с "ресурсами" Mac OS. Что между ними общее, в чем различия?
5. Каковы оригинальные свойства файловой системы BeOS?
6. Почему в файловой системе настольной BeOS обязательно должна обеспечиваться целостность?
7. Охарактеризуйте правила диспетчеризации нитей в BeOS. Классифицируйте их в соответствии с классификацией главы 2 части I.

Глава 10. Операционная система QNX

10.1. Архитектура

Операционная система QNX [32] производится компанией QNX Software Systems Ltd. с 1980 г. В настоящее время существует версия 6.1 системы. Для нас ОС QNX представляет особый интерес по двум причинам: во-первых, это ОС реального времени, во-вторых, это ОС, построенная на концепции микроядра "в чистейшем виде". Как следствие этого QNX – легко масштабируемая система

Архитектура ОС QNX показана на рисунке 10.1.



Рисунок 10.1 Архитектура ОС QNX.

Микроядро QNX имеет минимальный размер (всего 8 Кбайт), и в нем сосредоточены все операции, выполняемые в режиме ядра. Ядро не имеет процессов, его модули всегда выполняются в контексте процесса, их вызвавшего. Модули, сосредоточенные в микроядре, выполняют следующие основные функции:

- планирование и переключение процессов и управление реальной памятью (планировщик);
- первичную обработку прерываний и перенаправление их адресатам (редиректор прерываний);
- обеспечение связей между процессами (средства взаимодействия);
- сетевые взаимодействия (сетевой интерфейс).

Все эти функции аппаратно-зависимые и/или требуют высокой эффективности в реализации. Другие функции ОС обеспечиваются системными процессами-менеджерами, которые, однако, выполняются в пользовательском режиме и с точки зрения микроядра ничем не отличаются от процессов пользователей. Типичные конфигурации ОС QNX включают в себя следующие системные процессы:

- менеджер процессов (Proc);
- менеджер файловой системы (Fsys);

- менеджер устройств (Dev);
- сетевой менеджер (Net).

Микроядро QNX выполняет всего 57 системных вызовов, однако процессы-менеджеры ОС обеспечивают выполнение большого числа других системных вызовов, что позволило ОС QNX получить сертификат POSIX. Таким образом, ОС QNX может считаться Unix-подобной системой, хотя ее внутренняя структура далека от традиционной структуры ОС Unix.

10.2. Управление процессами

Порождение и планирование процессов обеспечивается менеджером процессов совместно с планировщиком в микроядре. Менеджер процессов выполняет порождение новых процессов, загрузку и завершение процессов. Для создания процессов имеются системные вызовы `fork()` и `exec()` – традиционные для Unix, а также `spawn()` – создание процесса-потомка с выполнением в нем новой программы.

QNX различает процессы, находящиеся в следующих состояниях:

- готовые (среди них – и активный процесс);
- ожидающие (6 подвидов – в зависимости от причин ожидания)
- мертвые (уже завершившиеся, но не передавшие информации о своем завершении).

Планирование процессов в QNX выполняется по абсолютным приоритетам, то есть появившийся или разблокированный процесс с более высоким приоритетом вытесняет активный процесс немедленно. При наличии в состоянии готовности нескольких процессов с одинаковым высшим приоритетом разделение процессора между ними выполняется по одной из дисциплин на выбор:

- FCFS;
- RR;
- динамическое изменение приоритета.

В последнем случае изменение приоритета производится по таким правилам:

- если активный процесс полностью исчерпал квант времени и есть еще процессы с таким же приоритетом, приоритет активного процесса уменьшается на 1;
- если процесс пробыл в очереди готовых процессов, не получая обслуживания на процессоре, 1 сек, его приоритет увеличивается на 1.

Всего в системе имеется 32 градации приоритетов.

В отличие от других систем, в которых процессы реального времени получают наивысший приоритет (в ущерб всем другим процессам), в QNX обеспечение работы в реальном времени состоит в том, что всем процессам обеспечивается высокая реактивность. То есть, если происходит какое-либо событие (прерывание), требующее выполнения определенного процесса, требуемый процесс становится активный после самой минимальной задержки. Реактивность обеспечивается за счет высокой реентерабельности модулей микроядра (то есть возможности прервать выполнение модуля) и высокой эффективности средств взаимодействия процессов.

Внешнее событие вызывает прерывание. Для обеспечения высокой реактивности прерывание должно обрабатываться немедленно. Но обработка прерывания может быть отложена по следующим причинам:

- выполняется обработка прерывания с более высоким приоритетом;

- выполняется нереентерабельный код микроядра (при этом обработка прерывания запрещается).

Если первый вид задержек является объективным и оправданным, то второй является нежелательным. В QNX модули микроядра тщательно оптимизированы с целью минимизации размера участков нереентерабельного кода. В результате модули микроядра также являются прерываемыми почти в любом месте. Участки кода с запрещенными прерываниями составляют в среднем всего 9 команд на входе в модуль микроядра и 14 команд – на выходе из модуля.

10.3. Средства взаимодействия

ОС QNX обеспечивает (на уровне микроядра) три средства взаимодействия процессов: сигналы, сообщения и поручения (проху).

Механизм сигналов соответствует тому, который мы рассмотрели в разделе 9.2 части I. QNX работает с большим количеством типов сигналов, среди которых:

- стандартные сигналы, определяемые POSIX;
- сигналы, управляющие работой процессов;
- специальные сигналы QNX;
- сигналы, поддерживающие старые версии Unix.

Процесс может определять способы обработки некоторых (но не всех) сигналов.

Сообщения являются основным способом взаимодействия между процессами в QNX. В отличие от того смысла, который мы вкладывали в этот термин в разделе 9.7 части I, в QNX сообщения являются синхронными, то есть процесс, пославший сообщение, требует обязательного ответа на него.

Обмен сообщениями обеспечиваются вызовами микроядра:

- `Send()` – посылка сообщения;
- `Recive()` – прием сообщения;
- `Reply()` – посылка ответа.

На рисунке 10.2 показан сценарий взаимодействия процессов при посылке сообщения

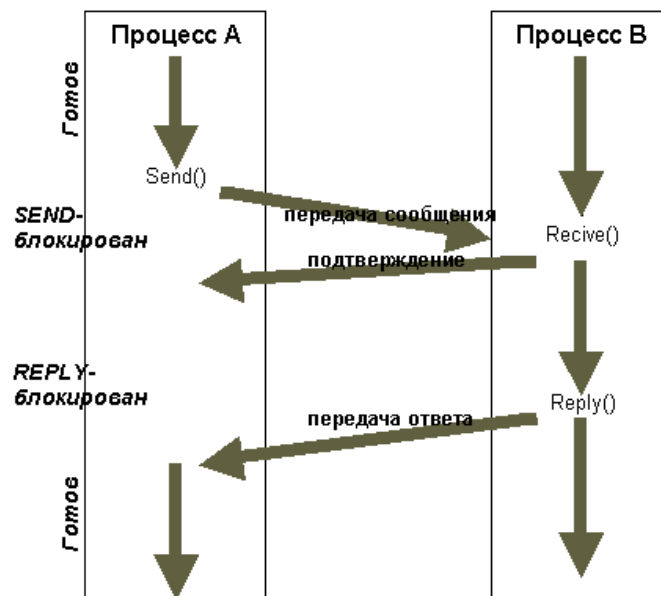


Рисунок 10.2 Сценарий взаимодействия процессов при посылке сообщения

В соответствии с протоколом передачи сообщений различаются следующие подвиды блокировки процесса:

- **SEND-блокированный процесс** – послал сообщение и ожидает подтверждения его приема.
- **REPLY-блокированный процесс** – получил подтверждение приема и ожидает получения ответа.
- **RECEIVE-блокированный процесс** – запросил прием сообщения и ожидает его поступления. На рисунке 10.2 состояние RECEIVE-блокировки не показано, в него мог бы попасть Процесс В, если

бы выполнил системный вызов `Recv()` прежде, чем Процесс А выполнил `Send()`.

Модель сообщений QNX более всего напоминает взаимодействие процессов по принципу рандеву (см. раздел 8.11 части I), описываемую как:

`A!x; ?y`

Для взаимодействия процессов необходима "встреча" готовности одного процесса (Процесса А) передать сообщение и готовности другого процесса (Процесса В) принять сообщение. При этом для процессов, участвующих в рандеву, нет необходимости знать о готовности процесса-корреспондента. Процесс, первым пришедший в точку рандеву, просто блокируется (SEND- или RECEIVE-блокировкой) до готовности процесса-партнера.

Передача ответа подтверждения не требует, и выполнение вызова `Reply()` не приводит к блокировке процесса, выполнившего этот вызов.

При необходимости процесс может посылать сообщения нулевой длины и/или ответы нулевой длины – такие приемы применяются для взаимного исключения и синхронизации без обмена данными.

Несколько процессов могут послать сообщения одновременно одному адресату. В этом случае сообщения могут обрабатываться (получаться адресатом) либо в порядке их поступления, либо в соответствии с приоритетами отправителей.

Еще один вызов микроядра – `Crecv()` – позволяет процессу проверить наличие сообщений для него и, таким образом, избежать RECEIVE-блокировки.

Интересно, что, используя механизм сообщений-рандеву, библиотеки системных вызовов QNX обеспечивают интерфейсы других стандартных средств взаимодействия процессов, таких как программные каналы или семафоры.

Третий вид взаимодействия – поручения – обеспечивает асинхронное взаимодействие процессов. Фактически поручения идентичны очередям сообщений, описанным нами в разделе 9.7 части I.

10.4. Файловая система

Администратор файловой системы ОС QNX позволяет стандартным образом организовать хранение и доступ к данным файловых подсистем (томов). С точки зрения логической файловой системы, хранение файлов в QNX подобно тому, какое обеспечивает ОС Unix: общее дерево каталогов с возможностью монтирования новых томов как ветвей этого общего дерева. Обеспечиваются также "жесткие" и "мягкие" связи.

На физическом уровне диск QNX структурирован так, как показано на рисунке 10.3.



Рисунок 10.3 Структура диска QNX

Загрузчик представляет собой блок начальной загрузки. Он не осуществляет загрузку собственно ОС, а выполняет выбор загрузочного файла ОС.

Корневой блок имеет структуру каталога и содержит информацию о следующих специальных файлах:

- файл с именем / – корневой каталог;
- файл с именем / .inodes – файл файловых индексов;
- файл с именем / .boot – загрузочный файл ОС;
- файл с именем / .altboot – альтернативный загрузочный файл ОС.

Загрузчик загружает операционную систему из файла / .boot, однако, имеется возможность загрузки и из альтернативного файла.

Битовая карта содержит информацию о свободных блоках на диске. Свободному блоку соответствует бит со значением 0, занятому – 1.

Размещение файла на диске QNX показано на рисунке 10.4.

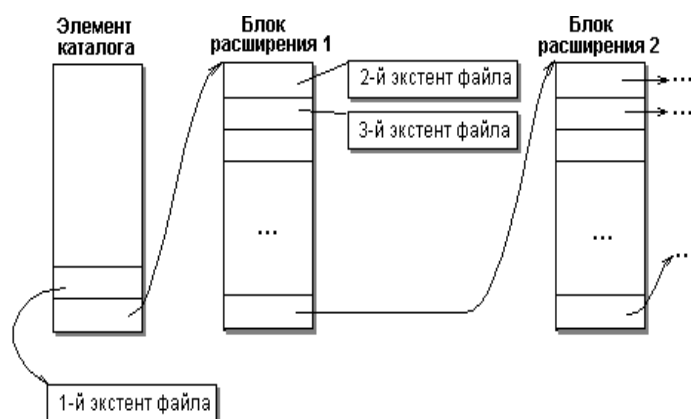


Рисунок 10.4 Размещение файла на диске QNX

Единицей распределения дискового пространства является блок (512 байт). Пространство файлу выделяется экстентами – непрерывными последовательностями, состоящими из целого числа блоков. В элементе каталога, соответствующего файлу, содержится номер начального блока и размер для первого экстенста файла. Если файлу выделено более одного экстенста, то в элементе каталога содержится номер блока расширения, через который адресуются следующие экстенсты файла. Если одного блока

расширения недостаточно, последний элемент первого блока расширения адресует второй блок расширения и т.д.

Интересным образом обеспечиваются в QNX жесткие связи (алиасы). Показанная на рисунке 10.4 структура относится к файлу, не имеющему жестких связей. Если же для файла создается жесткая связь, то информация о размещении файла переносится в файловый индекс, находящийся в файле `/.inodes`. Элемент каталога в этом случае содержит номер файлового индекса, и два разных элемента каталога могут ссылаться на один и тот же файловый индекс, а следовательно, на один и тот же физический файл. Таким образом, файловый индекс для файла создается только тогда, когда нужно разделить информацию о хранении файла в логической и в физической файловых системах.

Файловый индекс создается также и для файла с длинным именем. В обычном элементе каталога предусмотрено место для 16-символьного имени файла. Если длина имени файла превышает 16 символов, для файла создается файловый индекс и информация о размещении файла переносится в файловый индекс. При этом в элементе каталога освобождается место еще для 32 символов имени, таким образом, длина имени файла может достигать 48 символов.

10.5. Управление устройствами

Интерфейс между процессами и устройствами обеспечивается менеджером устройств. Устройства включены в общее пространство имен файловой системы как специальные файлы, находящиеся в подкаталоге `\.dev`. Для процесса устройство представляется как двунаправленный поток байтов. Менеджер устройств управляет прохождением этого потока между процессом и устройством и отчасти осуществляет обработку

данных в потоке. С каждым устройством связан блок управления `termios`, в котором задаются параметры обработки данных, такие как:

- алгоритм передачи данных (скорость, контроль четности и т.д.);
- отображение символов, вводимых с клавиатуры, на экране;
- трансляция вводимых символов;
- программное и аппаратное управление потоком данных;
- и т.д., и т.п.

Данные, которыми обмениваются менеджер устройств и драйвер проходят через набор очередей, с каждым устройством связаны по три очереди:

- входная очередь;
- выходная очередь;
- так называемая каноническая очередь – очередь ввода данных с редактированием.

Общий размер всех трех очередей не превышает 64 Кбайт. Очереди обслуживаются по дисциплине FIFO, независимо от приоритетов процессов, которым принадлежат данные в очередях. Для обеспечения высокой эффективности ввода-вывода сам менеджер устройств выполняется как процесс с высоким приоритетом. Это не сказывается на быстроте других процессов, так как управление вводом-выводом никогда не занимает процессор надолго. Драйверы также выполняются как отдельные процессы, их приоритеты зависят от особенностей обслуживаемых ими устройств.

Вводимые данные помещаются драйвером во входную очередь. Менеджер устройств выбирает данные из этой очереди только тогда, когда процесс запрашивает данные. Выходные же данные менеджер устройств помещает в выходную очередь и немедленно же активизирует драйвер.

Запрос данных при пустой входной очереди приводит к блокировке процесса. Также блокируется процесс и тогда, когда пытается вывести данные при переполненной выходной очереди.

10.6. Сетевые взаимодействия

С самого начала QNX создавалась как сетевая ОС и это выражается прежде всего в том, что средства взаимодействия локальных и удаленных процессов в QNX одни и те же – сообщения. Процесс не видит разницы во взаимодействии с локальным или удаленным корреспондентом. Такое свойство обеспечивается при помощи "виртуальных каналов", показанных на рисунке 10.5.

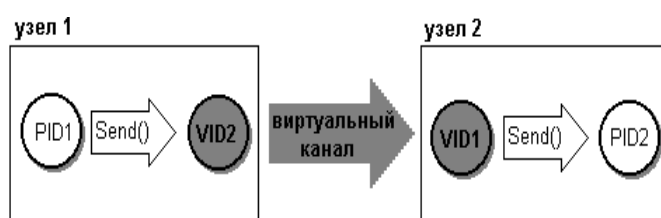


Рисунок 10.5 Посылка сообщения через виртуальный канал

Виртуальный канал создается процессом-отправителем сообщения. При этом на узле отправителя и на узле получателя создаются виртуальные процессы, каждый из которых представляет на локальном узле идентификатор процесса – удаленного корреспондента. Реальный процесс имеет реальный идентификатор (PID), виртуальный процесс – виртуальный идентификатор (VID). VID обеспечивает соединение, которое содержит следующую информацию:

- локальный PID;
- удаленный PID;
- удаленный NID (идентификатор узла сети);

- удаленный VID.

Процессы QNX имеют символьные имена, причем эти имена могут быть глобальными, доступными во всей сети. Приложение может по имени получить PID процесса – удаленного корреспондента. При этом система автоматически создает виртуальный канал и для приложения этот канал отождествляется с PID корреспондента.

Администратор сети обеспечивает создание виртуальных каналов, буферизацию данных в канале и контроль целостности виртуальных каналов.

10.7. Графическая система Photon

Пользовательский интерфейс QNX строится на базе графическая система Photon. Структура графической системы представляет для нас интерес прежде всего потому, что она следует общим архитектурным концепциям QNX. Это обстоятельство делает графическую систему нетребовательной к ресурсам, легко масштабируемой – от интерфейса встроеного или карманного мобильного устройства до полнофункционального WIMP-интерфейса. Это обеспечивает также то, что возможные сбои графической системы не оказывают влияния на работоспособность всей ОС и требуют только перезапуска отказавшего компонента.

В отличие от других графических систем, которые обеспечивают функции графического интерфейса в монолитной (Windows) или клиент/серверной (X Window) модели, Photon строится на базе компактного графического микроядра и распределения графической

функциональности между взаимодействующими процессами. Архитектура графической системы показана на рисунке 10.6, и она очень похожа на архитектуру QNX в целом.

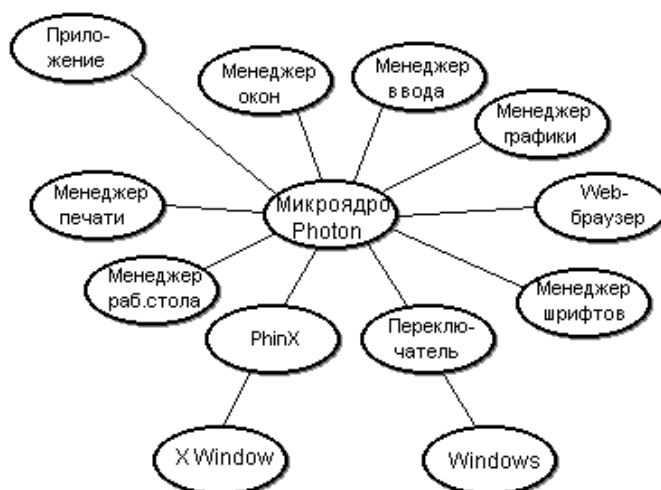


Рисунок 10.6 Архитектура графической системы Photon

Микроядро Photon, которое является процессом QNX, выполняет необходимый минимум графических функций. Микроядро Photon занимает всего 55 Кбайт памяти. Прочие части графической системы – также процессы, которые для выполнения базовых функций обращаются к микроядру, используя средства взаимодействия, обеспечиваемые микроядром QNX. Менеджеры графической системы являются опционными, включение новых менеджеров расширяет функции системы. До некоторой степени ключевым компонентом, определяющим переход от интерфейса встроенной системы к WIMP-интерфейсу, является менеджер окон, который обеспечивает изменение размера, минимизацию, перемещение и т.д. для окон.

Работа системы Photon строится на концепции "трехмерного пространства событий", которая иллюстрируется на рисунке 10.7.

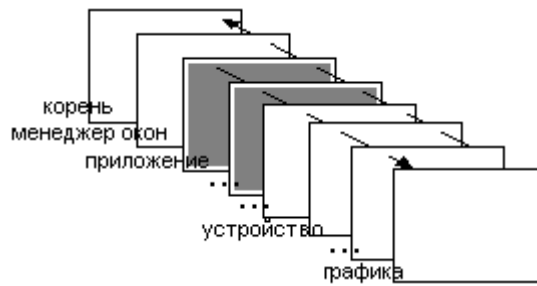


Рисунок 10.7. Движение через пространство событий

Событиями в системе являются как события, инициируемые пользователем (мышью, клавиатурой), так и события, инициируемые процессами. Пространство, через которое движутся события, представляется как набор параллельно размещенных прямоугольных областей. Метафорой, давшей название системе, является движение частицы света (фотона) через ряд стеклянных пластин. На одном конце этого ряда находится корневая область, создаваемая системой, на другом конце – та область, которая представляется пользователю. Процесс, который выполняет какие-либо функции, связанные с интерфейсом, помещает свою область. Каждая область имеет две для проходящих через нее события: маску чувствительности и маску непрозрачности. Установка бита чувствительности для определенного события определяет передачу события для обработки процессу, связанному с областью. Установка для события бита непрозрачности определяет прекращение движения события через пространство. Графические драйверы являются процессами, которые помещают свои области на переднем (ближнем к пользователю) краю ряда. Это обеспечивает также возможность распределенной обработки в сети: приложение с графическим интерфейсом может работать в одном узле сети, а результат его работы – отображаться на другом узле. Физический драйвер целевого узел просто помещает свою область на переднем краю пространства событий. Подобным образом обеспечивается и отображение результатов работы приложений QNX в других графических системах:

вместо драйвера в пространство событий вставляется область переходника в систему Windows или X Window.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Опишите архитектуру QNX. Какой архитектурной концепции она соответствует?
2. Каков обязательный минимум системного программного обеспечения для функционирования QNX?
3. В чем суть QNX как системы реального времени?
4. Какие дисциплины планирования процессов обеспечивает QNX?
5. Какие из описанных в Части I механизмов взаимодействия процессов используются в QNX?
6. Сопоставьте файловую систему QNX с различными файловыми системам Unix. Что между ними общего, в чем различия?
7. В чем состоит масштабируемость графической системы Photon?
8. Какие свойства QNX делают эту ОС пригодной для применения на тонких клиентах?

Глава 11. Вычислительная система AS/400

11.1. Архитектура

Семейство вычислительных систем AS/400 [11, 23] является результатом длительного эволюционного развития вычислительных систем IBM среднего класса. Основные архитектурные концепции, заложенные в этой системе, сформировались еще в IBM System/38 (1978 г.). Архитектура AS/400 представлена на рисунке 11.1.

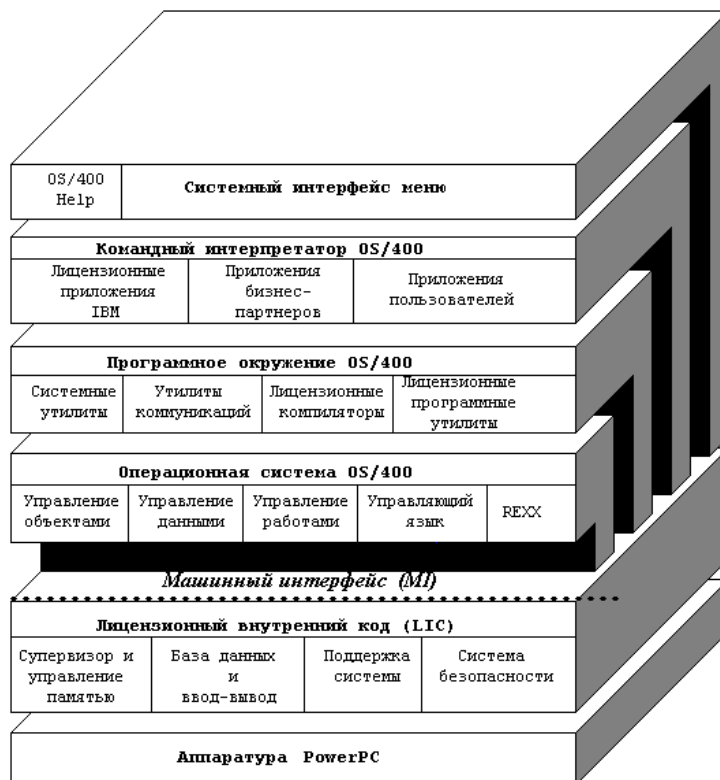


Рисунок 11.1 Архитектура AS/400

В архитектуре AS/400 сочетаются концепции иерархической архитектуры, архитектуры на базе микроядра, виртуальной машины и объектно-ориентированного подхода. Иерархическая структура программного обеспечения системы хорошо видна на рисунке. Самый нижний слой программного обеспечения выделен в так называемый Системный Лицензионный Внутренний Код (SLIC – System Licensed Internal Code) и составляет микроядро. Программное обеспечение слоя микроядра фирма не включает в ОС, считая его входящим в состав оборудования. Исторически сложилось так, что в самой первой системе, применяющей этот подход – IBM System/36 этот слой действительно был изолирован аппаратно (выполнялся на другом процессоре), далее в IBM System/38 и в ранних моделях AS/400, строившихся на базе 48-разрядного CISC-процессора, этот слой был реализован как программно, так и

микропрограммно. При переходе AS/400 на 64-разрядный RISC-процессор PowerPC (1995 г.) этот слой стал чисто программным.

Интерфейс микроядра – в AS/400 он называется MI (machine interface – машинный интерфейс) – обеспечивает функционально полную систему команд, в символьном виде представляемую высокоуровневым языком ассемблера. Таким образом, MI предоставляет лежащему выше программному обеспечению интерфейс некоторой виртуальной машины. И приложения, и сама ОС OS/400 разрабатываются на уровне MI (или выше), не имея доступа к интерфейсам, лежащим ниже MI, в том числе и к командам реального процессора. Переносимость программного обеспечения – приложений и ОС – обеспечивается на уровне MI-кодов. MI-код не является непосредственно исполняемым, он должен быть переведен в команды реального процессора. Однако, процесс трансляции расположен ниже уровня MI, он совершенно прозрачен для приложений и для ОС. Среди команд MI имеются как команды, близкие к обычным машинным командам, оперирующие байтами, словами, числами и т.п., так и команды, оперирующие с интегрированными структурами данных – объектами, обрабатываемыми микроядром. Впрочем, "обычные" команды MI также можно назвать объектно-ориентированными: команды содержат не собственно данные, а ссылки на объекты, содержащие наряду с самими данными и описания их типа, размера и т. п. Так, например, системы команд реальных процессоров обычно содержат несколько команд сложения – разных для разных размеров и форм представления чисел; в MI имеется единственная команда сложения – ADDN, которая в зависимости от типов операндов транслируется в ту или иную команду (или последовательность команд) реального процессора.

SLIC обеспечивает аппаратную независимость верхних уровней программного обеспечения – приложений и OS/400. При упомянутом переходе на платформу PowerPC переделке подвергся только SLIC,

операционная система и все приложения (более 20 тысяч приложений) не изменялись, но, как будет показано ниже, были автоматически оптимизированы для новой, 64-разрядной платформы.

AS/400 отличается значительной степенью системной интеграции и высоким уровнем системных интерфейсов. Ряд системных функций в AS/400 выполняются SLIC (лежат ниже уровня MI), ряд – OS/400, выполнение же большинства функций распределено между ОС и микроядром. Примеры такого распределения показаны на рисунке 11.2. Так, многие функции, традиционно выполняемые ОС, здесь реализованы на уровне SLIC (защита, ввод-вывод, управление памятью); многие функции, традиционно обеспечиваемые утилитами и отдельными приложениями, интегрированы в OS/400 (база данных, пользовательский интерфейс).

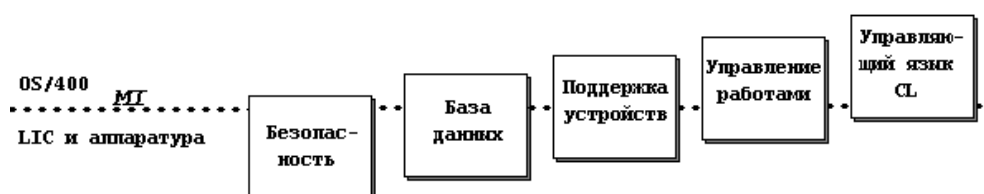


Рисунок 11.2 Распределение функций между OS/400 и SLIC

11.2. Объекты

Еще одной ключевой концепцией AS/400 является ее объектно-ориентированная структура. Выше мы упомянули об объектах MI в AS/400, для оперирования которыми имеются специальные команды MI В системе имеется predetermined набор типов объектов – системные объекты, часть из которых является объектами MI, часть – объектами OS/400. Некоторые OS/400-объекты отображаются в MI-объекты "один к

одному" (хотя иногда – под другими названиями), другие OS-объекты совершенно не связаны с объектами MI, некоторые OS/400-объекты представляют собой интеграцию MI-объектов. Примеры OS/400-объектов: программа, документ, таблица, файл, команда, профиль пользователя, библиотека (объект, содержащий другие объекты), папка (объект, содержащий другие объекты и папки). Примеры MI-объектов: программа (то же, что и в ОС), модуль, пространство данных, индекс, очередь, профиль пользователя (то же, что и в ОС), контекст (библиотека – в ОС). Следует отметить, что объекты нижнего уровня – объекты MI – доступны как для приложений, так и для OS/400 только через специальные команды MI и не могут обрабатываться каким-либо иным образом, минуя эти команды. Аналогично объекты OS/400 доступны для приложений только через системные вызовы.

Общие свойства системных объектов перечисляются ниже.

1. Объект, как правило, имеет имя. Объект может не иметь имени, если он не используется никем, кроме ОС.
2. Объект должен быть явным образом создан командой MI CREATE. Команда ссылается на созданный пользователем шаблон (template) объекта, который содержит атрибуты и значения объекта.
3. Объект может быть постоянным или временным – этот атрибут объекта определяется при его создании. Постоянные объекты сохраняются в системе до их явного уничтожения (командой MI DELETE). Временные сохраняются не дольше, чем до перезагрузки системы. При явном уничтожении объекта занимаемое ими адресное пространство очищается, но не освобождается, сохраняется также заголовок объекта, в котором, однако, делается пометка "уничтоженный". Это позволяет корректно обрабатывать ошибочные обращения к уже удаленным

объектам. Освобождение пространства и уничтожение заголовка происходит при перезагрузке системы.

4. Адресация к объектам производится через системные указатели. Управление правами доступа к объекту выполняется ниже уровня МІ (подробно рассматривается далее).
5. Временные объекты могут помещаться в "группы доступа". "Группа доступа" – системный объект, который позволяет связывать объекты различного типа и осуществлять к ним совместный доступ.
6. Атрибуты объекта могут быть "материализованы", то есть, скопированы в доступную область.

Объектам выделяется пространство, в котором располагаются данные и указатели. Внутреннее представление объекта занимает один или несколько сегментов памяти и включает в себя:

- заголовок объекта, содержащий его атрибуты (постоянный/временный, имя и тип объекта, описание занимаемого объектом пространства, ссылки на профили создателя и владельца, временные отметки и т.п.);
- функциональную часть, содержащую данные объекта, зависящие от его типа – это могут быть структуры данных, программные коды и т.п.; единственный тип МІ-объекта – "пространство" – не имеет функциональной части;
- область данных, содержащую пользовательские данные, она используется, как рабочая область при манипуляциях с объектом.

Функциональная часть объекта полностью инкапсулирована. Доступ к содержимому функциональной части возможен только через объектно-ориентированные команды МІ (команды материализации), в которых объект адресуется "системным указателем". Доступ к содержимому области данных возможен непосредственный, через "указатель

пространства" или "указатель данных". Область данных объекта может содержать любые данные, в том числе и системные указатели на другие объекты.

Следует отдельно упомянуть системный объект, называемый "машинный индекс", внутреннюю структуру которого составляет двоичное дерево. Такие объекты составляют основу всех системных таблиц и каталогов микроядра и ОС, а также входят в состав объектов встроенной реляционной базы данных (DB2 for AS/400). Именно это обстоятельство позволяет нам утверждать, что система управления базой данных в AS/400 глубоко интегрирована в систему: она использует те же средства, что и ОС и даже микроядро.

Системный указатель, адресующий объект, может быть "разрешенным" (resolved) или "неразрешенным" (unresolved). Неразрешенный системный указатель содержит символьную идентификацию объекта. Разрешенный системный указатель представляет собой 16-байтную структуру данных, в состав которой входит прямой адрес объекта в пространстве одноуровневой памяти. Оперировать с объектами можно только, используя разрешенный системный указатель. Для доступа к объекту системный указатель должен быть разрешен, что обеспечивается командой MI RESOLVE, схема действия которой показана на рисунке 11.3. Эта инструкция преобразует символьный указатель в разрешенную форму. Выполнение разрешения включает в себя:

- поиск объекта по имени;
- проверка соответствия типа запрашиваемого доступа типу объекта;
- проверка (по пользовательскому профилю) прав доступа данного пользователя к данному объекту;
- проверка захвата объекта другим процессом.

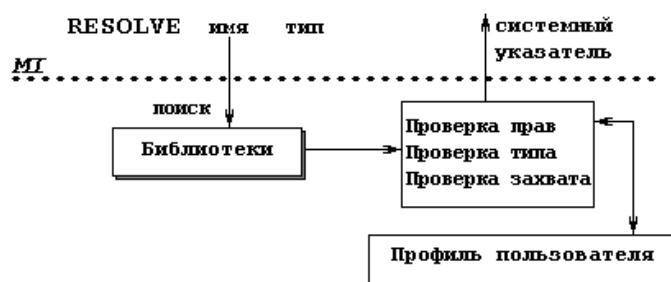


Рисунок 11.3 Разрешение системного указателя

11.3. Управление памятью

Одноуровневая модель памяти AS/400 описана в разделе 3.8 части I. Все управление памятью расположено ниже уровня MI и обеспечивается не ОС, а "виртуальным оборудованием" – микроядром SLIC. Механизмы, одноуровневой модели памяти, таким образом, реализуются на аппаратном и самом низком программном уровне и являются прозрачными не только для процессов пользователей, но и для ОС. Интерфейс же управления памятью (или то, что ему здесь соответствует) используется одинаковым образом и процессами пользователей, и OS/400. Виртуальная память выделяется сегментами (в старых моделях размер сегмента составлял 64 Кбайт, в новых – 16 Мбайт). Любой объект занимает целое число сегментов. Запрос на выделение памяти приводит к созданию нового объекта "область данных". Для совместимости с другими ОС в последних версиях системы добавлена возможность выделения для процесса отдельного частного адресного пространства размером до 1 Тбайт. Всего возможно создавать до миллиона терабайтных адресных пространств.

Некоторые аппаратные средства процессора PowerPC поддерживают защиту памяти в одноуровневой модели. Во-первых, дескриптор страницы, как и во многих других процессорах, содержит бит защиты от записи и бит пользователь/система. Эти биты защищают страницы, занятые

системными объектами. Еще один механизм аппаратной защиты включен в процессор Power PC специально для AS/400. В системе различаются несколько типов указателей, из которых для нас представляют интерес прежде всего "системные указатели" и "указатели памяти". Последние являются указателями в привычном для нас смысле, они могут модифицироваться, например, операциями адресной арифметики, но только в пределах установленного диапазона для объекта "область памяти". Системные указатели представляют адреса объектов. Их создание и работа с ними возможны только специальными командами MI. Процесс не должен модифицировать системные указатели, но если системные указатели расположены в области данных, то они не застрахованы от модификации их процессом. Механизм ярлыков контролирует модификацию системных указателей. Ярлыком (tag) является дополнительный бит, связанный с каждым машинным словом и индицирующий модификацию этого слова. В качестве ярлыка используется один из битов корректирующего кода для каждого 64-разрядного слова. Доработки, внесенные в процессор Power PC для адаптации его к AS/400, в значительной степени состояли во включении в систему команд процессора команд, работающих с tag-битом. Эти команды, однако, недоступны на уровне MI, следовательно, tag-бит для уровня OS/400 и приложений прозрачен. При корректных (через инструкции MI) операциях с системным указателем в составе соответствующих инструкций MI выполняются команды, устанавливающие tag-биты указателя в 1, любые другие команды модификации данных сбрасывают tag-биты в 0. При использовании данных в качестве системных указателей проверяются ярлыки двух 64-битных слов, составляющих системный указатель, если хотя бы один из них сброшен, генерируется прерывание-ловушка. Таким образом, описанный механизм не защищает системные указатель от изменений, но

предотвращает неправильное использование системных указателей. При вытеснении страницы на внешнюю память ярлыки всех слов страницы группируются в отдельную структуру, которая записывается в заголовок сектора на дисковой памяти.

11.4. Программы и процессы

Подготовка программ в системе AS/400 также имеет некоторые интересные особенности, представленные на рисунке 11.4. в несколько упрощенном виде.

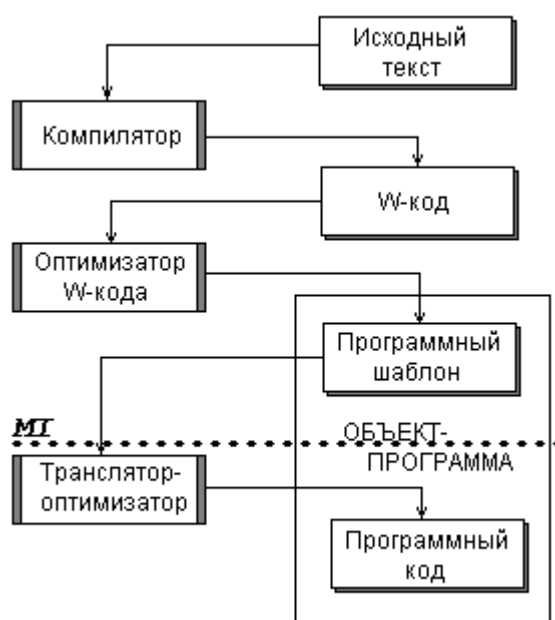


Рисунок 11.4 Подготовка программ в системе AS/400

Компилятор языка высокого уровня (C, PL/1 и т.д.) выполняет трансляцию исходного текста в промежуточный W-код, общий для всех блочных языков. Над промежуточным кодом выполняется системно-независимая оптимизация, а затем программа транслируется в MI-представление, называемое программным шаблоном. Программный шаблон является частью объекта "программа" (это объект как MI, так и OS/400). При запуске программы на выполнение происходит обращение к

SLIC. SLIC проверяет наличие в программном объекте кодовой части и, если таковая отсутствует, выполняет трансляцию шаблона в систему команд конкретного процессора. Транслятор во внутримашинное представление, выполняющий также и машинно-зависимую оптимизацию, входит, таким образом, в состав SLIC. Программные коды сохраняются в составе программного объекта, и при последующих запусках на той же вычислительной системе трансляция не выполняется. Поскольку объект "программа" инкапсулирован, составные части объекта (шаблон и коды) доступны только через определенные для объекта операции. Для объекта "программа" не существуют операции "читать" и "писать", что исключает возможность модификации программного кода, например, вирусом. При переносе программного объекта в другую вычислительную систему переносимой частью является программный шаблон. При первом запуске на новой системе SLIC определяет, что программные коды не соответствуют новой платформе и перетранслирует шаблон. Поскольку при такой трансляции выполняется аппаратно-зависимая оптимизация, внутреннее представление получается максимально эффективным именно для той модели процессора, на которой оно будет выполняться. Такое свойство системы позволило ее разработчикам при резкой смене архитектуры процессора не только сохранить все имеющиеся на текущий момент приложения, но и обеспечить без дополнительных затрат их полноценный перевод на 64-разрядную архитектуру. В целях экономии пространства программный шаблон может быть удален из состава программного объекта, но это ограничит переносимость программы на другие платформы.

Схема подготовки программ для AS/400, показанная на рисунке 11.4, как мы отметили, неполная. На самом деле выходе транслятора-оптимизатора трансляции получается объект "модуль". Этот объект затем обрабатывается компоновщиком, также входящим в состав SLIC, и

результатом является кодовая часть объекта-программы. В системе различаются "программы", имеющие единственную входную точку, и "системные программы", наборы процедур, используемые как библиотеки статической или динамической компоновки. В языке МП имеется два вида команд вызова: "вызов процедуры" и "вызов программы". Первый используется для обращения к модулям, включенным компоновщиком в состав программного объекта и статически привязанным к программе. Второй – устанавливает связь с другими модулями динамически, при выполнении вызова. Проблемы размещения здесь не возникает, поскольку динамически подключаемые модули, как и все прочие объекты уже размещены в одноуровневом адресном пространстве. Проблема прав доступа решается на общесистемном уровне защиты объектов.

Единицей работы в системе является процесс или задача. В системе различаются четыре состояния задач: "подвешенность" (задача начинается или завершается), готовность, исполнение, ожидание. Блок контекста задачи представляется весьма масштабной структурой данной, которая обрабатывается исключительно микроядром. С каждой задачей связан Блок диспетчеризации задачи. Блоки диспетчеризации всех задач, готовых к выполнению, объединены в очередь диспетчеризации задач и упорядочены в ней по приоритетам. Первый Блок диспетчеризации (или первые n блоков в n -процессорной конфигурации) относится к активной (исполняемой) задаче. Приоритеты задач перевычисляются динамически по оригинальному алгоритму "планировщика с оценкой задержки", который позволяет оптимизировать загрузку в многопроцессорных конфигурациях. Допускается, однако, и выполнение задач вне порядка их приоритетов. Наиболее важной составляющей процесса является так называемая группа активизации, которая, прежде всего, предоставляет процессу ресурсы памяти – статическую память, стек, кучу. По умолчанию процесс имеет две группы активизации – системную и пользовательскую.

Понятием более высокого уровня является задание. Задание – единица работы с точки зрения пользователя, выполняемая одним или несколькими процессами. Задания выполняются в подсистемах, каждая из которых представляет собой предопределенную среду, предназначенную для выполнения в ней однотипных заданий (например, пакетных, интерактивных и т.д.). Подсистеме выделяется некоторый набор ресурсов, таким образом, подсистемы полностью изолированы друг от друга.

Интересно рассмотреть, как в AS/400 поддерживается работа с нитями. Теоретически можно было бы обеспечить выполнение нитей в составе процесса, выделяя каждой нити свою группу активизации. Однако такая структура нитей повлекла бы за собой необходимость слишком серьезных изменений в микроядре. Для обеспечения совместимости со стандартами разработчики прибегли к паллиативу. Нить представляется в виде отдельной задачи, которая разделяет с другими задачами-нитьями одну и ту же группу активизации. Для обеспечения требования быстрого создания нитей в системе при загрузке создается пул "пустых" задач – с уже выделенными для них блоками контекста и некоторыми наиболее общими ресурсами. При необходимости запуска задачи-нити из этого пула выбирается пустая задача и формируются ее индивидуальные ресурсы. Такое решение нельзя не признать остроумным, однако, сами разработчики признают его недостаточно эффективным и рассматривают только как временную меру до реализации поддержки нитей в микроядре.

11.5. Постоянные объекты и ввод-вывод

Взаимодействие между процессами в AS/400 обеспечивается через очереди сообщений, передачу которых обеспечивает SLIC. Очереди поддерживаются "классическими" семафорами.

Как мы отметили выше, одноуровневая модель памяти замещает понятие "файл" понятием "постоянный объект". (Собственно "файлами" в AS/400 называются файлы встроенной СУБД.) Базовая логическая структура хранения объектов в AS/400 – двухуровневая: объекты размещаются в библиотеках. Для идентификации объекта следует указать имя библиотеки, имя объекта и тип объекта (объекты разного типа могут иметь одинаковые имена). Единственная системная библиотека – QSYS – может включать в себя другие библиотеки. Для обеспечения функций продукта Office Vision в AS/400 был включен тип объекта "папка" – контейнерный объект с неограниченным уровнем вложенности. Этот объект послужил также основой продукта PC Support/400, обеспечивающего хранение в AS/400 файлов ПЭВМ, работающих под управлением MS DOS или OS/2. В дальнейшем для обеспечения полного набора клиент/серверных возможностей с широким разнообразием типов клиентов был создан новый продукт – Client Access/400 и Интегрированная файловая система IFS (Integrated file system). IFS позволяет представить структуру хранения всех объектов в AS/400 в виде иерархической файловой системы клиента (MS DOS, OS/2, Windows, Unix, Nowell, NFS и т.д.). При этом подкаталоги верхнего уровня соответствуют различным файловым системам клиентов. В основе IFS лежит базовая двухуровневая система хранения, которая является подмножеством любой иерархической файловой системы, верхние уровни над ней строятся при помощи объектов-папок.

Неотъемлемой частью архитектуры AS/400 являются процессоры ввода-вывода. Взаимодействие приложения с процессором ввода-вывода происходит через механизм обмена сообщениями, поддерживаемый SLIC. Данные могут передаваться как в составе сообщения, так и выбираться процессором ввода-вывода непосредственно из оперативной памяти. Сообщение – запрос ввода-вывода проходит через ряд последовательных

ступеней обработки в OS/400 и SLIC, на каждой ступени к обработке привлекаются соответствующие объекты операционной системы и микроядра, описывающие логические и физические устройства. В одной из возможных конфигураций AS/400 в качестве процессора ввода-вывода применяется процессор Intel/Pentium, работающий под управлением OS/2 Warp или Windows NT. Вычислительная система, таким образом, может выполнять функции сетевого файлового сервера, не загружая этой работой свой системный (центральный) процессор.

Отметим, что диски не входят в эту систему управления устройствами. В соответствии с концепцией одноуровневой памяти, управление дисками относится к управлению памятью и полностью обеспечивается в SLIC.

11.6. Система безопасности

Одной из наиболее интересных особенностей AS/400 является ее система безопасности, реализованная в микроядре SLIC. Как мы показали, команда RESOLVE осуществляет контроль доступа к объекту, причем контроль этот выполняется в микроядре, ниже уровня MI. Разрешенный системный указатель является константой – он не может быть изменен в программе. Системные указатели, которые находятся в области данных какого-либо объекта, могут быть изменены (по ошибке или при попытке "взлома" системы), но теги защищают систему от использования модифицированных системных указателей. Концепция безопасности AS/400 в настоящее время включает в себя 5 возможных уровней защиты. Уровень выбирается при конфигурировании системы. Краткие характеристики уровней приводятся ниже.

Уровень 10. Отсутствие защиты.

Уровень 20. Защита входа в систему паролем. Дополнительным средством этого уровня является возможность регистрации "пользователя с ограниченными возможностями", для которого определяется допустимое подмножество команд.

Уровень 30. Защита ресурсов. К предыдущему уровню добавляется контроль прав доступа пользователей к объектам – системным и пользовательским.

Уровень 40. Защита ОС. К предыдущему уровню добавляется защита от нестандартного доступа к объектам. Этот уровень был введен для того, чтобы пресечь создание независимыми разработчиками приложений, использующих внутреннюю структуру объектов: такие приложения оказывались непереносимыми на следующие версии AS/400. Для обеспечения этого уровня были определены два состояния системы: системное и пользовательское, и некоторая часть команд МІ стала привилегированной. В МІ было введено огромное число новых команд для корректного доступа к объектам.

Уровень 50. Защита по С2. К предыдущему уровню добавляется контроль событий, связанных с безопасностью.

Дальнейшее рассмотрение мы ведем на уровнях 40 и 50.

AS/400 в основном использует защиту, ориентированную на списки возможностей. Ниже описывается информация по защите, содержащаяся в профиле пользователя.

1. Имя пользователя и пароль для входа в систему.
2. Класс пользователя. Различаются такие классы:
 - Офицер безопасности – имеет полный набор прав доступа ко всем объектам;
 - Системный администратор – устанавливает права доступа для Пользователей рабочих станций;

- Системный программист, разработчик приложений, имеет права доступа к инструментальным средствам разработки приложений;
 - Системный оператор – имеет доступ к средствам обслуживания системы, системным очередям, утилитам и т.д.;
 - Пользователь рабочей станции, пользователь, работающий в среде определенного приложения – имеет ограниченный доступ к библиотекам используемого им программного продукта.
3. Объекты, к которым пользователь имеет доступ. В профиле имеются два списка таких объектов: тех, для которых пользователь является владельцем, и тех, для которых у пользователя определены частные права. Создатель объекта становится его владельцем, но может передать право владения другому пользователю. Владелец может устанавливать для других пользователей частные права на объект.
4. Права доступа – определяются для каждого объекта в вышеуказанных списках. Различаются следующие права доступа:
- операционное – получать описание объекта и пользоваться другими правами;
 - управления – определять защиту для объекта, перемещать и переименовывать объект;
 - существования – удалять объект, выполнять его сохранение и восстановление, передавать право владения;
 - управления правами – устанавливать частные права для других пользователей;
 - группа прав данных – читать, добавлять, удалять, изменять записи в объекте (каждое из этих действий составляет отдельное право);
 - имеются также коды для стандартных комбинаций прав, а также "право" EXCLUDE – отмена всех прав, в том числе и прав,

определяемых групповыми профилями, и публичных прав (см. ниже).

Для каждого объекта все возможные права доступа кодируются в одном элементе списка.

5. Специальные права – определяют подмножество возможностей пользователя, применимых ко всем объектам в системе, например: полный доступ ко всем объектам, доступ ко всем пользовательским профилям, управление любыми заданиями в системе и т.д.

Списки возможностей являются основными информационными структурами защиты, но в системе поддерживаются некоторые дополнительные структуры, позволяющие сократить списки возможностей. Списки доступа – создаются для группы объектов, по отношению к которым все права одинаковы. Список доступа состоит собственно из двух списков – списка объектов и списка пользователей с их правами. Права разных пользователей могут быть различны, но права одного пользователя одинаковы по отношению ко всем объектам. В списке возможностей в профиле пользователя перечисление объектов может быть заменено идентификатором списка доступа, в который входит данный пользователь и данные объекты. Некоторые объекты содержат в своих дескрипторах также коды доступа, определяющие публичные права – права, действительные для всех пользователей.

Пользователи могут объединяться в группы с одинаковым доступом к объектам. Для группы создается групповой профиль, содержащий список возможностей группы.

В числе атрибутов объекта "программа" имеется один, позволяющий закрепить за программой "адаптированные" права – права ее создателя, эти права, сохраняются только на время выполнения программы.

Алгоритм авторизации (поиска прав доступа), выполняемый при разрешении системных, указателей состоит из следующих шагов.

1. Поиск индивидуальных прав, выполняемый в такой последовательности:
 - специальные права в пользовательском профиле;
 - права и частные права в пользовательском профиле;
 - права по списку доступа (если объект входит в такой список).
2. Поиск групповых прав в такой последовательности:
 - специальные права в групповом профиле;
 - частные права в групповом профиле;
 - групповые права по списку прав доступа.
3. Поиск публичных прав:
 - публичные права в коде доступа объекта;
 - публичные права по списку доступа.

Права, определенные в различных информационных структурах, не суммируются, поиск прекращается, как только найдено первое право для объекта. Таким образом, права индивидуальные имеют приоритет перед групповыми, а групповые – перед публичными. Индивидуальные права могут определять большие возможности пользователя, чем групповые, а могут – и меньшие. Если в индивидуальные права включается код EXCLUDE, то поиск заканчивается отказом в доступе. Исключения составляют адаптированные права, которые добавляются к найденным по вышеприведенному алгоритму.

11.7. Новые свойства и перспективы

Начиная с версии OS/400 V4R4 в AS/400, была введена архитектура логических разделов (LPAR), которая обеспечивает представление одной вычислительной системы как нескольких систем – каждая с собственным процессором/процессорами, памятью, устройствами ввода-вывода и со

своей ОС. Разделы функционируют независимо друг от друга и логически отделены от других разделов. Коммуникации между разделами осуществляются через операции ввода-вывода (виртуальный Ethernet). Архитектура программного обеспечения AS/400 с LPAR показана на рисунке 11.5.



Рисунок 11.5 AS/400 с логическими разделами

Для управления логическими разделами нужен гипервизор, то есть некоторый слой системного программного обеспечения, который будет выполнять общее управление всеми ОС в разделах. В функции гипервизора входит:

- распределение ресурсов между разделами;
- инсталляция ОС в разделах;
- запуск и останов ОС в разделах;
- обеспечение взаимодействия между разделами
- и т.д.

Для выполнения этих функций гипервизор должен иметь собственные службы, например, управления памятью, синхронизации и т.п.

В системное программное обеспечение AS/400 введен еще один слой – PLIC (Partitioning Licensed Internal Code). Некоторые функции управления разделами реализованы в самом PLIC, для выполнения других

PLIC работает совместно с частью SLIC одного из разделов, называемого первичным. Таким образом, гипервизор состоит из PLIC и части SLIC первичного раздела.

Первичный раздел обязательно должен быть в системе, и только один раздел может быть первичным, остальные разделы – вторичные. AS/400 продается как система только с первичным разделом. Этот раздел является владельцем всех ресурсов вычислительной системы. Далее покупатель может из первичного раздела создавать вторичные разделы, распределять и перераспределять ресурсы между ними, устанавливать в них ОС. Некоторые ресурсы (например, сервисный процессор) могут принадлежать только первичному разделу. Из первичного раздела также происходит взаимодействие оператора со всей системой в части управления разделами.

Каждому разделу выделяется собственное подмножество физических ресурсов – процессор или процессоры, память, средства ввода-вывода и т.д. В первой реализации LPAR в V4R4 каждому разделу требовался один физический процессор (или более). В V5R1 допускается логическое разделение (квантование процессоров). Объем ресурсов процессора и памяти, выделенный разделу, может быть изменен только при перезагрузке раздела. Ресурсы же, базирующиеся на процессорах ввода-вывода, могут перераспределяться между разделами динамически.

Обычно ОС использует области, расположенные по некоторым фиксированным адресам физической памяти, для хранения каких-то системных управляющих структур, обработки прерываний и т.п. Поскольку в LPAR физическая память распределена между разделами, это становится невозможным для ОС раздела. Поэтому все программное обеспечение раздела, включая SLIC, является перемещаемым. Первичную обработку прерываний выполняет PLIC, но его функции здесь сводятся к

нахождению вектора прерываний соответствующего SLIC и передаче управления на него.

Структура системного программного обеспечения AS/400 такова, что над SLIC может располагаться не обязательно OS/400, а и какая-то другая ОС. Такая возможность была использована еще в самых первых версиях системы: AS/400 обеспечивает работу также ОС своих предшественниц – System/36 и System/38. Однако реализация над SLIC "совсем другой" ОС осуществилась только в V5R1 в 2001 г. И такой ОС стала ОС Linux. Первая версия – это 32-битный Linux для PowerPC на базе ядра 2.4, но уже в 2002 г. будет реализована и 64-битная версия. Linux устанавливается во вторичном логическом разделе AS/400 и требует всего 0,1 процессора и 64 Мбайт памяти на раздел, поэтому на одной вычислительной системе можно иметь десятки инсталляций Linux. В первичном разделе, однако, должна работать OS/400. OS/400 обеспечивает для Linux-разделов виртуальные устройства ввода-вывода, доступ к системным службам OS/400 и взаимодействие с приложениями, выполняющимися в OS/400.

Некоторое время вычислительные системы среднего класса были "нелюбимым детищем" фирмы IBM и не слишком усердно продвигались на рынке. Однако, с 1995 года ситуация кардинально меняется. AS/400 попадает в число стратегических продуктов фирмы, объемы ее продаж резко возрастают. Все программные продукты IBM оперативно портируются на эту вычислительную систему, а в ряде случаев она становится первой платформой, на которой эти продукты внедряются. AS/400 сейчас, безусловно, находится на подъеме, как в отношении своих позиций на рынке, так и в отношении развития системы. Среди постоянно развивающихся свойств AS/400 следует прежде всего отметить ее движение к Открытым Системам. AS/400 поддерживает до 90% спецификаций Single Unix Specification, и в нее включены все API Unix для бизнес-вычислений, в том числе: файловый ввод-вывод, средства

взаимодействия между процессами, функции управление процессами, сигналы, нити. интерпретатор shell и т.д.

Еще одно направление развития интероперабельности системы – обеспечение интеграции служб OS/400 со службами ОС Windows 2000, которая может выполняться на одном из процессоров ввода-вывода AS/400.

В настоящее время AS/400 или iServer представляет собой семейство вычислительных систем с широкими возможностями масштабирования (старшие модели семейства отличаются по производительности от младших более чем в 300 раз), работающих под управлением OS/400 v.4.5 или v.5.1. Дальнейшее развитие и эскалацию на рынке этой системы можно прогнозировать на длительный срок.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие архитектурные концепции реализованы в программном обеспечении AS/400?
2. Программы, разработанные в конце 70-х годов для System/36 и System/38, выполняются на современных моделях AS/400, полностью используя их возможности. Какие архитектурные особенности AS/400 обеспечили такую переносимость?
3. Сопоставьте объектно-ориентированную архитектуру AS/400 с объектно-ориентированной архитектурой Windows NT/2000. Какая система более полно воплощает объектный подход?
4. Что дает основание утверждать, что реляционная СУБД DB2 встроена в AS/400?
5. В чем принципиальное отличие концепции памяти AS/400 от всех других рассмотренных нами систем?

6. Как AS/400 поддерживает работу с нитями? Сопоставьте нити AS/400 с нитями Open Unix и с нитями BSD Unix.
7. Какие из описанных в главе 6 части I методов управления устройствами применяются в AS/400?
8. Что представляет собой программа в AS/400? Почему программы AS/400 неуязвимы для компьютерных вирусов?
9. Какие уровни защиты могут быть установлены для AS/400? Какими средствами обеспечивается аудит доступа на уровне 50?
10. Какими средствами защиты памяти поддерживается система безопасности AS/400?
11. Опишите сценарий процесса авторизации для AS/400.
12. Сопоставьте структуры хранения информации о правах в AS/400 и в Windows NT/2000. Что между ними общего, в чем различия?
13. Какие изменения в архитектуре системного программного обеспечения AS/400 повлекло за собой введение логических разделов?

Глава 12. Операционные системы мейнфреймов

12.1. История и архитектура мейнфреймов

Обычно на русский язык термин "мейнфрейм" переводится как "большая ЭВМ универсального назначения". Однако нам представляется, что в настоящее время такое определение уже не является точным. Современные мейнфреймы не универсальны. Они специализированы как компьютеры для обработки больших и сверхбольших объемов данных или

как суперсерверы. Название одного из поколений мейнфреймов IBM ESA (Enterprise System Architecture – архитектура систем масштаба предприятия) достаточно точно отражает такую специализацию

Мейнфреймы фирмы IBM [7] имеют почти 40-летнюю историю развития, причем, развитие это протекало эволюционно, во всяком случае, с точки зрения пользователей. При любых изменениях в аппаратной архитектуре каждое следующее поколение мейнфреймов обеспечивало выполнение программного обеспечения, разработанного для предшествующих поколений, почти в полном объеме.

За время существования мейнфреймов неоднократно высказывались и приобретали широкую популярность заявления об их устаревании и скорой кончине, однако, всегда "эти слухи оказывались несколько преувеличенными", и мейнфреймы продолжали существовать и развиваться. В настоящее время в технологиях обработки информации возрастает потребность в существенно централизованных решениях, и новое поколение мейнфреймов оказывается востребованным, как никогда раньше.

Семейство мейнфреймов IBM System/360, появившееся в начале 60-х годов, стало значительной вехой в истории вычислительной техники. Во-первых, это были первые ЭВМ, которые начали выпускаться серийно, а не по индивидуальным проектам, во-вторых, они стали первым семейством ЭВМ, то есть набором моделей с разной производительностью и разной стоимостью, но с переносимостью программного обеспечения с одной модели на другую. Семейство IBM System/360 строилось на базе CISC-процессоров с богатым набором команд и несколькими режимами адресации. Эти процессоры, однако, не поддерживали динамическую трансляцию адресов, поэтому программное обеспечение работало с реальной памятью, привязка адресов осуществлялась при загрузке. (Точнее – во время выполнения, в момент загрузки "базовых" регистров, но

загруженная в реальную память программа уже не могла быть перемещена.) Размер адресной шины составлял 24 бита, что позволяло адресовать 16 Мбайт памяти – реальной и виртуальной. Чрезвычайно сильным свойством IBM System/360 явилась архитектура каналов ввода-вывода [8] (см. главу 6 части I). Достоинства мейнфреймов IBM System/360 определили ведущее положение этого семейства на рынке вычислительной техники в течение всех 60-х и начала 70-х годов, и первое время конкуренты IBM, вынуждены были делать собственные компьютеры программно совместимыми с IBM System/360.

Следующим поколением мейнфреймов стало семейство IBM System/370. Принципиальным отличием его от предыдущего поколения явилось введение динамической трансляции адресов. Применялась сегментно-страничная модель трансляции, во всех ОС этого поколения каждому процессу выделялся один сегмент адресного пространства (АП), то есть процесс обладал собственной виртуальной памятью размером в 16 Мбайт. Однако в этом поколении проявилось некоторая "успокоенность" фирмы IBM. Фирма упустила из виду одно из конкурирующих направлений развития вычислительной техники, а именно – мини-ЭВМ, так называемые Unix-машины, ведущим производителем которых в то время была фирма Digital Equipment. Нововведений семейства IBM System/370 оказалось недостаточно, чтобы сохранить почти монопольное положение на рынке, и именно тогда возникла первая "легенда о смерти мейнфреймов".

Отличие семейства IBM System/370/XA (eXtended Architecture – расширенная архитектура) от предыдущего поколения было достаточно революционным: адресная шина расширилась до 31 бита, что позволило адресовать виртуальную память до 2 Гбайт (при этом сохранилась совместимость и со старыми 24-разрядными моделями). Другим принципиально важным нововведением расширенной архитектуры

явилось введение в подсистему ввода-вывода возможности динамического определения пути к устройствам ввода-вывода и поддержка SMP-архитектуры.

Следующим поколением стало семейство IBM ESA/370. В этом семействе появилась возможность адресовать до 16 2-Гбайтных виртуальных АП. Важнейшим из других возможностей, по-видимому, явилось свойство PR/SM (Partition Resources/System Management), обеспечивающее возможность разбиения (на микропрограммном уровне) ресурсов вычислительной системы на независимые логические разделы. Семейства 370/XA и ESA/370 определили новую специализацию мейнфреймов, однако еще не вывели фирму IBM в абсолютные лидеры.

Дальнейшее развитие мейнфреймов происходило во многом благодаря конкуренции IBM с японскими фирмами (Hitachi, Fujitsu), выпускающими собственные мейнфреймы, программно совместимые с IBM. Новое семейство – IBM ESA/390 интегрировало в себе большое количество нововведений, которые в итоге определили "второе рождение" мейнфреймов. Среди этих нововведений – увеличение регистрового массива, новые средства защиты памяти, новые средства работы с числами с плавающей точкой, оптоволоконные ESCON-каналы, встроенные криптографические процессоры и аппаратная поддержка сжатия данных и, конечно, sysplex – средство комплексирования вычислительных систем. В этом семействе произошел также переход мейнфреймов на CMOS-технологии, что привело к тому, что по размерам и по энергопотреблению они стали сравнимы даже с ПЭВМ.

Семейство ESA/390 прочно восстановило позиции мейнфреймов в мире информационных технологий, но дальнейшее развитие требований к обработке данных повлекло за собой и появление нового семейства мейнфреймов – z/900 [40]. Главная особенность новой архитектуры – расширение адресной шины до 64 разрядов. Для понимания

функционирования программного обеспечения и ОС мейнфреймов мы приведем некоторые минимальные сведения об аппаратной части z-архитектуры. На рисунке 12.1 показана логическая структура современного мейнфрейма, так называемого z-сервера.

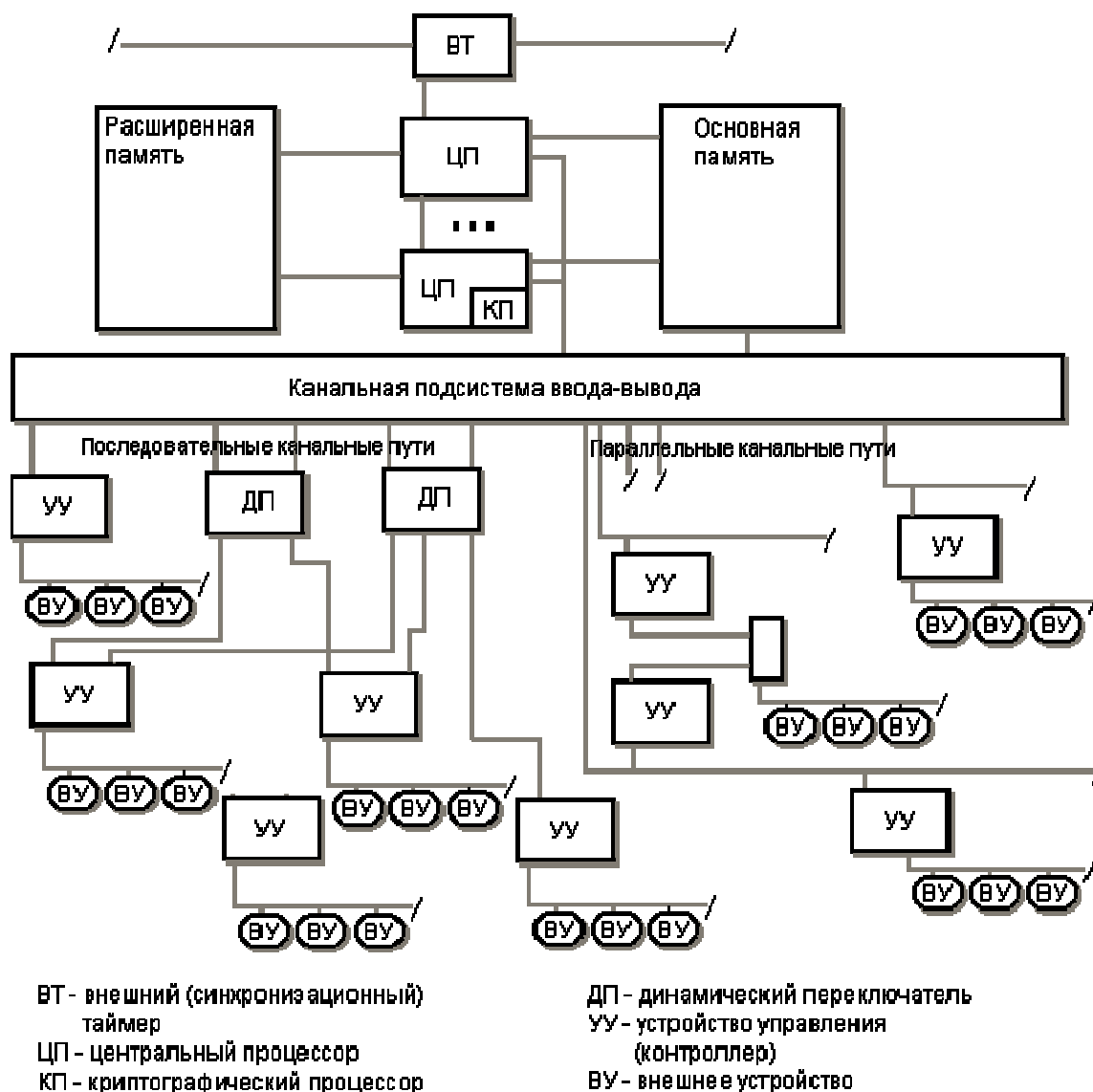


Рисунок 12.1 Логическая структура z-сервера

Основные вычислительные свойства реализуются на симметричной многопроцессорной (до 16 z-процессоров) конфигурации. Однако реально процессоров может быть и больше, так как конфигурация может включать в себя помимо основных z-процессоров специализированные сервисные процессоры, криптографические процессоры, процессоры ввода-вывода и

т.д. Z-процессор, как и все предыдущие поколения центральных процессоров мейнфреймов, является CISC-процессором. Свойства CISC используются в этой архитектуре в полной мере. Обязательной составной частью z-архитектуры является Лицензионный Внутренний Код (LIC), реализованный на уровне микропрограмм процессора. Интенсивное использование микропрограммирования позволяет включить в систему команд процессора очень мощные команды, обеспечивающие значительную поддержку работы операционных систем и даже конкретных приложений. Одно из различий в моделях z-процессоров состоит в том, реализованы те или иные команды в них аппаратно (в более производительных моделях) или микропрограммно.

Основной аппаратной структурой, в которой фиксируется состояние процессора, является 16-байтное Слово Состояния Программы (PSW – Program State Word). В нем отражается адрес выполняемой команды, состояние задача/супервизор, режим адресации и т.п. Дополнительная информация о состоянии содержится еще в 16 8-байтных управляющих регистрах. В системе имеется 16 8-байтных регистров общего назначения (пара смежных таких регистров может использоваться для представления 16-байтного значения) и 16 16-байтных регистров плавающей точки.

Система имеет основную (оперативную) и расширенную память. Команды и обрабатываемые данные находятся в оперативной памяти. Расширенная память является необязательным компонентом системы. Она используется как дополнительный буфер между оперативной и внешней памятью. Данные могут перемещаться между основной и расширенной памятью постранично – командами PAGE IN и PAGE OUT.

В z-процессоре адрес имеет размер 64 бита, что позволяет работать с адресным пространством (АП) размером 16 эксабайт, однако процессор поддерживает и "старые" режимы адресации – с 31-битным и 24-битным

адресом (режим определяется состоянием соответствующих разрядов PSW).

В системе адресации различаются адреса: абсолютные, реальные и виртуальные адреса нескольких типов.

Абсолютный адрес – адрес в реальной памяти, фактический адрес ячейки памяти.

Реальный адрес, как правило, совпадает с абсолютным, кроме реальных адресов, меньших 8 Кбайт. Реальный адрес, меньший 8 Кбайт, преобразуется в абсолютный путем префиксации – добавления к нему значения, записанного в префиксном регистре. Область реальной памяти до 8 Кбайт используется для специальных целей системой прерываний и ввода-вывода, префиксация обеспечивает для каждого процессора в многопроцессорной системе собственную область младших адресов памяти.

Виртуальные адреса различаются четырех типов: первичные, вторичные, домашние и определяемые регистрами доступа. Для виртуальных адресов разного типа по-разному выполняется динамическая трансляция адреса. Режим динамической трансляции задается определенными битами PSW и управляющих регистров. В зависимости от режима в процессе динамической трансляции адресов используются от двух до пяти управляющих таблиц переадресации (3 таблицы областей, таблица сегментов, таблица страниц). В системе имеется также 16 AR-регистров (регистры доступа). Регистр AR0 содержит указатель на таблицы переадресации для первичного АП. Регистры AR1-AR15 позволяют приложению адресовать еще 15 дополнительных АП.

Защита памяти в мейнфреймах z-архитектуры, включает в себя традиционную для многих компьютерных систем изоляцию АП виртуальной памяти, бит защиты от выборки, бит обращения и бит изменения в дескрипторах страниц, а также предусматривает механизм,

основанный на применении 7-разрядных ключей памяти. Такой ключ приписывается каждой 4-килобайтной странице. В дескрипторе каждого страничного кадра имеется 4-битный ключ доступа, обеспечивающий авторизацию программ при обращении к памяти. Каждая программа имеет свой 4-битный ключ доступа, который при выполнении программы заносится в определенные разряды PSW. При каждом обращении к памяти ключ защиты, который выбирается из PSW, сравнивается с ключом страницы, к которой происходит обращение. Запись разрешается, только при совпадении ключей защиты. Системные (привилегированные) программы выполняются с нулевым ключом защиты, что дает им доступ к любой странице памяти.

Система ввода-вывода основывается на каналах ввода-вывода, описанных нами в главе 6 части I. Однако там мы описали строго иерархическое подключение "канал – контроллер – устройство", которое применялось в ранних реализациях. Современная архитектура мейнфреймов обеспечивает более сложную схему подключений с гибким установлением путей к устройству. Канальная подсистема ввода-вывода управляет потоком данных между основной памятью и устройствами. Как часть операции ввода-вывода канальная подсистема выполняет проверку доступности канальных путей, выбор одного из доступных путей и инициализацию операции обмена. В системе имеется два типа канальных путей:

- Параллельные канальные пути, служащие для поддержки интерфейса ввода-вывода System/360 и System/370; такой путь представляет собой электрические проводные соединения между канальной подсистемой и одним или несколькими контроллерами. До 8 контроллеров и до 256 устройств могут использовать совместно один параллельный путь.

- Последовательные каналные пути ESCON и FICON состоят из двух волоконно-оптических кабелей, динамических переключателей и контроллеров. Динамическое переключение может быть выполнено между двумя любыми последовательными каналными путями в этой же или в другой каналной подсистеме. К каждому контроллеру последовательного интерфейса может быть подключено до 256 внешних устройств.

Внешний таймер (ETR – external time reference) обеспечивает синхронизацию часов мейнфреймов, объединенных в тесно связанный комплекс (Parallel Sysplex).

Аппаратные средства z-архитектуры поддерживают программное обеспечение всех предыдущих архитектур мейнфреймов IBM, аналогично и ОС мейнфреймов развиваются эволюционным путем [21]. Эта эволюция происходит по трем параллельным линиям, история которых представлена на рисунке 12.2.

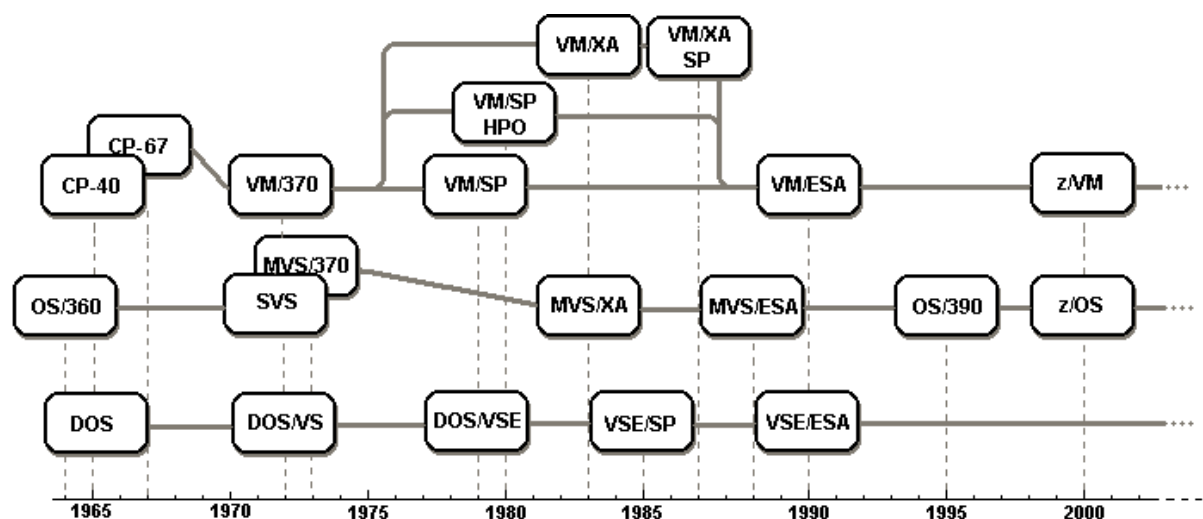


Рисунок 12.2 Эволюция ОС мейнфреймов

12.2. Операционная система VSE/ESA

Линия ОС, представляемая сегодня VSE/ESA v.2.6 [21, 24, 38], ориентирована на применение на младших, наименее мощных моделях мейнфреймов. Поэтому ей свойственны более простые решения, запаздывающее внедрение новых свойств аппаратной платформы (в частности, она пока не использует новых возможностей z-архитектуры), отсутствие развитых средств управления производительностью. Хотя имеется много примеров успешного построения промышленных информационных систем на базе VSE, ее основное назначение – поддерживать "унаследованное" программное обеспечение, разработанное для предшествовавших версий аппаратуры и ОС. Программисту, воспитанному на ПЭВМ, это может показаться странным, но в сфере промышленной обработки данных достаточно широко применяется программное обеспечение, разработанное 20 и более лет назад. За столь длительный срок эти программы доказали свою полезность и надежность, и у пользователей нет оснований от них отказываться.

Среда выполнения, которую VSE обеспечивает для приложений, показана на рисунке 12.3. Эта среда обеспечивается отчасти обязательными компонентами в составе ОС, отчасти – опционными компонентами ОС, отчасти – промежуточным программным обеспечением. Ниже вкратце рассматриваются компоненты, создающие эту среду.

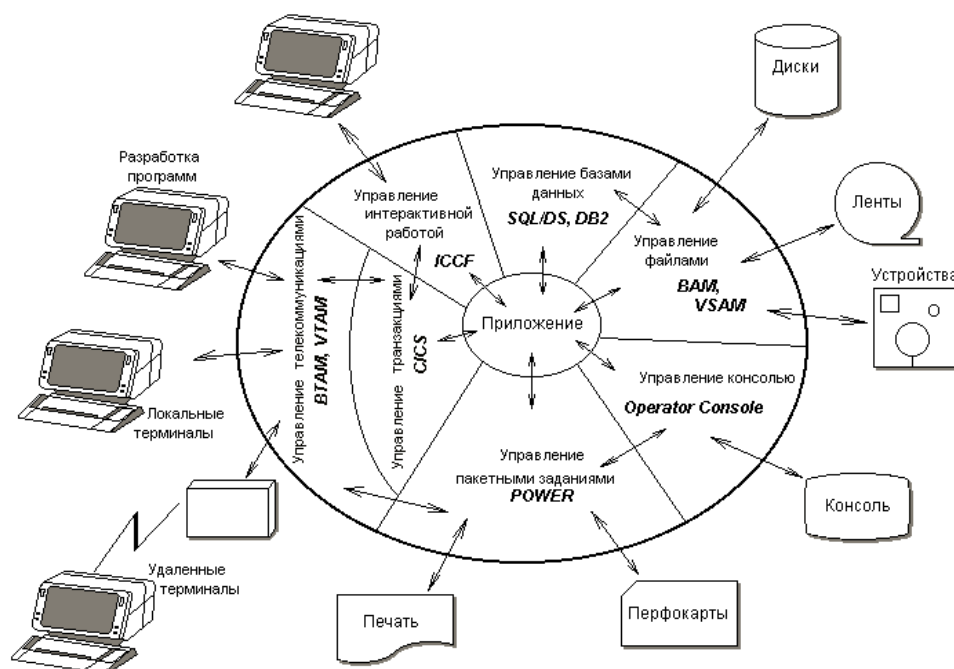


Рисунок 12.3 Среда выполнения приложения в VSE/ESA

Базовые управляющие средства обеспечиваются обязательным компонентом ОС, который носит название VSE/AF (Advanced Functions). В состав этого компонента входят: ядро ОС – супервизор, обеспечивающее управление памятью, управление задачами (в терминологии IBM задача означает процесс), базовые функции управления заданиями и базовые функции управления файлами, некоторые системные утилиты и т.д.

Управление памятью

Аббревиатура VSE расшифровывается как Virtual Storage Extension – расширение виртуальной памяти. Это название сложилось исторически, но сейчас его нельзя считать вполне точным. Первая ОС этой линии – DOS – работала только с реальной памятью. Реальная память разбивалась на разделы фиксированного размера, и в каждом разделе выполнялась одна задача. В DOS/VSE за счет динамической трансляции адреса System/370 создавалось виртуальное АП размером 16 Мбайт, которое затем разбивалось на разделы фиксированного размера – и для такой модели

название VSE является вполне справедливым. Однако, уже в VSE/SP и далее – в VSE/ESA появилась возможность создавать для каждого раздела независимое АП размером 16 Мбайт (а позже – 2 Гбайт). Структура памяти для современных версий VSE представлена на рисунке 12.4.



Рисунок 12.4 Структура памяти VSE/ESA

Нижняя часть виртуальной памяти – общая для всех разделов, в ней размещается ядро ОС – супервизор. Выше располагается совместно используемая область виртуальной памяти, которая содержит программы и данные, доступные для всех разделов. Другая совместно используемая область находится в верхней части 2-Гбайтного АП. Разделение совместно используемой области на две части связано с переходом от 24-разрядного адреса к 31-разрядному. Унаследованные программы с 24-разрядной адресацией видят только часть АП до 16 Мбайт, в которой есть все, что им

нужно. Программы же созданные с учетом 31-разрядной адресации видят все 2 Гбайт АП, в том числе и расширение разделяемой области.

Для части или для всего АП задачи может быть определен режим GETVIS, задающий расположение этой части в реальной памяти и исключающий ее из страничного обмена. Разумеется, это снижает эффективность функционирования остальных задач и применяется только для системных задач.

При старте системы автоматически создается 12 статических разделов. В системе есть также возможность создавать и динамические разделы (до 200 разделов) – такие разделы создаются автоматически при запуске задачи в области динамических разделов и уничтожаются при ее завершении. Кроме того, 31-разрядным приложениям ОС может предоставлять (через регистры AR) 2-Гбайтные "пространства данных", которые могут использоваться для хранения данных. В этих пространствах также могут создаваться виртуальные диски.

В части статических разделов, как правило, уже при старте системы запускаются системные задачи. Так, в разделе 0, который называется BG, обычно запускается задача связи с оператором – BG; в разделе 1 – задача POWER и т.д. Разделы этих задач, выполняющих обязательное общесистемное обслуживание, обычно (но не обязательно) размещаются в общей части АП, как показано на рисунке 12.4.

Управление задачами

Единицей работы в ОС является задание (job). Задание состоит в последовательном выполнении нескольких шагов-задач (task) – программ (в частном случае задание может состоять из единственного шага). Задание характеризуется классом (буква) и приоритетом (число). Для каждого раздела оператором задаются классы заданий, выполняемых в разделе и приоритет класса в разделе. Задания одного класса выбираются на

выполнение в соответствии с числовым приоритетом, а при равенстве приоритетов – в порядке поступления. Классы и приоритеты заданий определяют порядок, в котором задания выбираются на выполнение, но не дисциплину распределения процессорного времени.

С точки зрения распределения процессорного времени, VSE является системой без деления времени, с абсолютными приоритетами. Вытесняющая многозадачность здесь реализована в том отношении, что задача с более высоким приоритетом, придя в состояние готовности, немедленно вытесняет с центрального процессора задачу с низким приоритетом. Приоритет задачи определяется номером раздела, в котором она выполняется. Наивысший приоритет имеет раздел 0, далее – раздел 1 и т.д., задачи в динамических разделах имеют самый низкий приоритет. Для эффективного использования многозадачных свойств VSE следует в статических разделах с меньшими номерами запускать обменные задачи.

При работе VSE/ESA на многопроцессорной конфигурации только один процессор в каждый момент времени может выполнять код в режиме супервизора (привилегированном режиме).

Задания в VSE/ESA бывают двух видов – пакетные и интерактивные. Базовые средства VSE/AF обеспечивают обработку пакетных заданий. Пакетное задание представляет собой набор операторов языка управления заданием JCL на перфокартах (виртуальных). Основными операторами языка JCL являются:

- // JOB – оператор заголовка задания;
- // OPTION – оператор установки параметров/режимов выполнения задания;
- // EXEC – шаг задания, вызов на выполнение программы;
- // ASSIGN – назначение физического устройства логическому файлу программы для шага задания;

- // DLBL – назначение физического дискового файла логическому файлу программы для шага задания;
- // EXEC PROC – выполнение процедуры в шаге задания; процедура представляет собой хранимый в библиотеке набор операторов JCL; процедура может иметь параметры и содержать некоторую логику (ветвление в зависимости от значений параметров и результатов выполнения отдельных шагов).

Данные могут включаться в пакет или выбираться из файлов и библиотек.

Обязательным компонентом VSE является VSE/POWER – подсистема управления входными и выходными и выходными очередями. POWER обычно запускается в разделе F1 и располагается в реальной памяти. POWER выполняет следующее:

- читает задания из различных источников и записывает их во входную очередь, располагающуюся на диске (очередь RDR);
- выбирает задания из очереди RDR (в соответствии с их параметрами) в соответствующие разделы и инициирует их выполнение;
- записывает выходные данные приложений в очереди LST (печать) и PUN (вывод на перфокарты);
- также в соответствии с параметрами заданий передает данные из выходных очередей на реальные устройства (перфокарточные устройства не используются в современных мейнфреймах, и данные, выведенные на перфокарты, остаются электронными, в таком виде они могут быть перенаправлены, например, во входную очередь);
- для сетевой среды POWER создает также очередь XMT для передачи данных между узлами сети.

Таким образом, POWER является системой спулинга, обеспечивающей разделение процессов ввода, обработки и вывода и параллельное выполнение этих процессов.

Описанные выше классы и приоритеты заданий относятся к входной очереди, RDR. Данные, выводимые в выходные очереди, также имеют классы и приоритеты, задаваемые независимо от входных. VSE/POWER имеет собственный управляющий язык JECL (Job Entry Control Language), основное назначение операторов которого – определение классов и приоритетов данных в очередях.

Файловая система

Сочетание структуры файлов на внешней памяти и способов обработки файлов в программе составляет метод доступа. В VSE/ESA применяются две группы методов доступа: базисные методы – BAM, "унаследованные" от старых версий и виртуальный последовательный метод – VSAM (применяемый также и в z/OS как единственная для этой ОС структура файловой системы). Обычно при инсталляции VSE создаются два дисковых тома. На этих томах устанавливаются системные файлы и библиотеки, но также остается место и для пользовательских файлов. Первичное управление дисковым пространством выполняется средствами BAM. На каждом диске выделяется пространство – область VSAM. С точки зрения BAM, вся эта область представляется как один файл, но внутри этого файла средства VSAM обеспечивают собственное управление дисковым пространством и создание VSAM-файлов.

В начале каждого диска находится метка тома (VOL1), содержащая имя тома и указатель на размещение оглавления тома. Оглавление тома – структура VTOC – содержит информацию о размещении на томе BAM-файлов. Средства BAM фактически переключают управление дисковым пространством на программиста: при создании файла программист должен

явным образом указать физический адрес файла на диске и его размер. Это выполняется средствами языка управления заданиями: после оператора // DLBL, относящегося к создаваемому файлу должен следовать один или несколько операторов // EXTENT, задающих адреса и размеры участков файла. ВМ-файл располагается в одном или нескольких (до 16) непрерывных участках дискового пространства. Дисковое пространство выделяется сразу при создании файла и не может быть перераспределено в дальнейшем. Элемент VTOC для каждого файла содержит его имя и до 16 пар "адрес–размер" – для каждого участка. Утилита VTOC помогает программисту вести карту распределения дискового пространства.

Основные файлы ВМ, создаваемые на диске DOSRES при инсталляции системы:

- системная библиотека IJSYRS.SYSLIB, необходимая для начальной загрузки системы;
- область страничного обмена;
- область очередей POWER;
- области файлов ICCF, CICS и других системных программ;
- каталог VSAM;
- область VSAM.

Часть системных библиотек и файлов инсталлируется в области VSAM.

Информация обо всех VSAM-файлах на диске сохраняется в каталоге VSAM. Каталог VSAM должен быть на каждом томе, содержащем область VSAM.

Для файлов VSAM дисковое пространство выделяется динамически, и файл может занимать несмежные участки дискового пространства. Пространство выделяется блоками фиксированного размера (размер выбирается), план размещения файла представляет собой В⁺-дерево. Кроме

того, "листья" дерева связаны в линейный список, что позволяет осуществлять быстрый последовательный доступ к данным файла. VSAM поддерживает физические структуры файлов четырех типов:

- ESDS (entry-sequenced data set) – неупорядоченные записи фиксированной или переменной длины;
- KSDS (key-sequenced data set) – записи фиксированной или переменной длины, упорядоченные по ключам;
- RRDS (relative-record data set) – записи фиксированной длины, упорядоченные по номерам;
- VRDS (variable-length relative-record data set) – записи фиксированной или переменной длины, упорядоченные по номерам.

Физическая структура файлов ESDS очевидна. Для файлов RRDS память выделяется сразу для всех записей файла, и относительная позиция записи вычисляется. В RRDS-файле могут быть "пустые места" – для записей, еще не занесенных в файл. Для файлов KSDS и VRDS строится индекс (B^+ -дерево с линейным списком листьев) ключей или номеров соответственно. Для этих файлов возможно создавать также любое количество альтернативных индексов – по любым другим ключам, альтернативный индекс ссылается на основной индекс. Хотя физическая структура файлов в VSE – записеориентированная, системный API предоставляет как записе-, так и байториентированный интерфейс.

Логическое структурирование хранения информации и в BАМ, и в VSAM основывается на концепции библиотек. Библиотека является контейнерным объектом, содержащим одну или несколько подбиблиотек. Подбиблиотеки содержат разделы (файлы). Память выделяется для библиотеки, библиотеки BАМ не могут увеличиваться в размерах сверх выделенного им пространства. Память для подбиблиотек выделяется динамически в пределах пространства библиотеки. Обычно подбиблиотеки

объединяют в себе данные одного определенного типа – исходные, объектные или загрузочные модули. Системная утилита LIBR обеспечивает операции по обслуживанию библиотек.

ICCF

Наряду с пакетными заданиями, в VSE есть возможность и интерактивной работы. Она обеспечивается компонентом VSE/ICCF (Interactive Computing Control Facility). ICCF не является строго обязательным компонентом ОС, но применяется практически при всех ее инсталляциях. ICCF выполняется в отдельном статическом разделе и обеспечивает пользователю терминала следующие возможности:

- ввод, просмотр и редактирование программ, заданий и данных;
- запуск с терминала заданий – интерактивных или пакетных в POWER;
- ведение библиотек ICCF (см. ниже);
- доступ к файлам VSE;
- доступ к очередям;
- интерактивное выполнение системных утилит;
- организацию и выполнение потока заданий в интерактивном разделе.

Для взаимодействия с пользователем ICCF использует несколько типов полноэкранных панелей:

- панели выбора (меню);
- панели ввода данных;
- списковые панели;
- панели подсказок;
- панель текстового редактора.

ISCF обеспечивает собственные библиотеки и подбиблиотеки, предназначенные прежде всего для хранения текстов программ и заданий. Файлы в библиотеках ISCF состоят из записей размером 88 байт, из которых первые 80 используются для данных, а в 8 байтах находятся два указателя, связывающие записи файла в двунаправленный список. Для библиотек ISCF определяются права доступа. С точки зрения доступа имеется три типа библиотек:

- COMMON – библиотеки, содержащие некоторую общую информацию (общие процедуры, макросы и т.п.), к таким библиотекам имеют доступ все пользователи, но только для чтения, только системный администратор имеет доступ к этим библиотекам для записи;
- PUBLIC – библиотеки, доступные всем пользователям для чтения и для записи;
- PRIVATE – библиотеки, доступные только для одного пользователя.

Права доступа назначаются системным администратором.

Другие компоненты

Для обеспечения одновременной работы многих пользователей и ряда других своих функций ISCF использует компонент VSE/CICS (Customer Information Control System), который обязательно должен устанавливаться вместе с ISCF.

Функциональность CICS значительно шире, чем только поддержка интерактивного интерфейса VSE. CICS является мощным сервером транзакций, который доступен на всех аппаратных и операционных платформах IBM и применяется для обеспечения совместного доступа к данным множества разнoplatformенных компонентов информационной

системы. VSE/CICS представляет собой набор программных единиц и системных таблиц, которые выполняются в отдельном статическом разделе и обеспечивают:

- безопасность – авторизацию доступа пользователей к данным;
- управление терминалами;
- управление задачами – с мультипрограммным управлением транзакциями в разделе CICS;
- управление программами, включая поддержку множественных языковых сред и параллельное выполнение транзакций в одной программе;
- сериализацию доступа к данным параллельно выполняющихся транзакций;
- ведение журнала и восстановление целостности данных после сбоев.

Во всех промышленных применениях VSE CICS является практически обязательным компонентом, и прикладные программы для таких применений создаются с использованием платформенно-независимого API CICS для доступа к данным.

VSE/VTAM (Basic Telecommunication Access Method) является базовым телекоммуникационным методом доступа, обеспечивающим управление локальными и удаленными устройствами с использованием протоколов BSC или Asynch. VTAM не требует отдельной инсталляции и входит в состав VSE/AF.

VSE/VTAM (Virtual Telecommunication Access Method) является дополнительным методом телекоммуникации, управляющим взаимодействиями между устройствами в сети SNA. Он поддерживает локальные и удаленные рабочие станции в одно- или многомашинной сети. Если VSE/ESA установлена в узле сети, VTAM позволяет:

- пользователям и приложениям получать доступ к приложениям, установленным в другой системе;
- обмениваться данными с другими системами;
- другим системам получать доступ к VSE.

VTAM устанавливается как опционный компонент VSE/ESA и выполняется как задача в отдельном статическом разделе.

12.3. Операционная система z/OS

z/OS (раньше – OS/390, еще раньше – MVS) является стратегической для IBM ОС мейнфреймов [21, 24, 41]. Именно в этой ОС в первую очередь осваиваются новые свойства аппаратной платформы, именно в этой ОС в первую очередь становятся доступными новые версии стратегических продуктов промежуточного программного обеспечения, именно эта ОС рассчитана на применение в самых мощных и производительных вычислительных комплексах и sysplex'ах (тесно связанных многомашинных комплексах, которые "выглядят" с точки зрения управления и распределения нагрузки как одна вычислительная система). Последняя на сегодняшний день версия этой ОС – z/OS V1R3.

ОС OS/360 MVT, находившаяся "у истоков" этой линии, работала только с реальной памятью, создавая в ней динамические разделы по мере необходимости. В ОС MVS сложились концепции управления виртуальной памятью и другими основными ресурсами, оставшиеся в принципе неизменными и до настоящего времени. Переименование системы в OS/390 было связано с интеграцией в систему ряда программных серверов, ранее существовавших в виде отдельных программных продуктов, а в z/OS – с адаптацией к 64-разрядной z-архитектуре. Длительная история эволюционного развития MVS – OS/390 – z/OS привела к тому, что на

сегодняшний день z/OS является системой настолько сложной и богатой возможностями, что описать их все даже на структурном уровне – задача невыполнимая в объеме одной книги. Тем не менее, мы попытаемся (ни в коей мере не претендуя на полноту) дать читателю некоторое представление о компонентах управления теми ресурсами, которые являются предметом нашего основного внимания.

Примерная структура системного программного обеспечения в составе z/OS показана на рисунке 12.5.



Рисунок 12.5 Структура программного обеспечения z/OS

Мы отметили, что развитие этой ОС происходило исключительно эволюционным путем. Внедрение новых возможностей управления ресурсами в ОС происходит, как правило, по следующему сценарию:

1. новый управляющий сервис разрабатывается и внедряется как отдельный программный продукт, продаваемый отдельно от ОС;
2. новый программный продукт включается в комплект поставки ОС;
3. новый продукт интегрируется с ядром ОС, возможно, становится частью ядра.

Хотя понятие "ядро" для z/OS точно не определено, мы называем ядром Базовую Управляющую Программу (BCP – Base Control Program), осуществляющую низкоуровневое управление такими ресурсами, как память, процессы, средства коммуникаций. Надстройки над низкоуровневым управлением (в составе самой BCP или на более высоких уровнях системного программного обеспечения) позволяют управлять политиками распределения ресурсов. Ряд системных сервисов, не входящих в состав ядра, но работающих в режиме супервизора, являются подсистемами – средами выполнения приложений. Дополнительные системные сервисы расширяют возможности сервисов, включенных в базовый комплект. Некоторые программные продукты IBM, относящиеся к классу промежуточного программного обеспечения, также можно назвать подсистемами, так как они создают собственные среды. Эти продукты также тесно интегрированы с системой, и в ядро системы включены функции поддержки этих продуктов.

Управление памятью

Управление памятью является, возможно, самым интересным свойством z/OS. Аббревиатура первого названия ОС – MVS расшифровывается как Multiply Virtual Storage и отражает именно аспект управления памятью. Каждая задача в MVS (и в ее современных наследниках) обладает собственным виртуальным АП. Размер этого АП составлял 16 Мбайт в ранних версиях ОС (24-битный адрес), 2 Гбайта, начиная с MVS/XA (31-битный адрес) и 16 эксабайт в z/OS (64-битный адрес). Мы рассмотрим сначала первые две модели адресации, а затем отдельно расскажем об "освоении" системой 64-битного адреса.

Распределение виртуального АП для 24- и 31-битного размера адреса показано на рисунке 12.6. Нижняя часть виртуального АП занята системой, она перекрывается для всех АП, но для прикладных программ недоступна.

Верхняя часть виртуального 16-Мбайтного АП – общая область памяти, занимаемая объектами, совместно используемыми разными задачами. Это как разделяемые объекты данных, так и совместно используемые программные коды, например, системные сервисные службы, такие как TSO и т.п. АП между этими двумя областями является частным АП задачи. При расширении АП до 2 Гбайт дополнительная часть общей области памяти, смежная со "старой" появляется по другую сторону 16-Мбайтной границы, остальная часть дополнительного АП является дополнительным частным пространством задачи. Таким образом, задачи, в которых выполняются программы, разработанные для 24-разрядных версий MVS, видят привычную для себя структуру 16-Мбайтного АП, задачи, созданные для новых версий, видят полную структуру 2-Гбайтного АП. Размещение в памяти и выполнение программы определяется параметрами RMODE и AMODE. Первый из этих параметров определяет размещение программы в нижней или верхней части АП. Значение параметра AMODE отображается на соответствующий бит PSW и определяет режим выполнения некоторых команд процессора, при AMODE=24 команды, работающие с адресами, используют 24-битный адрес, при AMODE=31 – 31-битный адрес. Каждая программная секция характеризуется своими параметрами RMODE и AMODE, таким образом, режимы адресации могут изменяться и в ходе выполнения одной задачи.

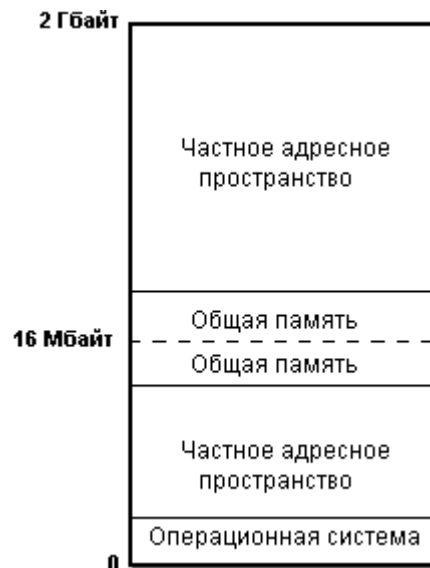


Рисунок 12.6 Виртуальное адресное пространство в OS/390

z/OS предоставляет также приложениям возможности использовать дополнительные АП. Хотя реализации всех этих возможностей используют описанные выше регистры доступа AR, с точки зрения приложений их можно разделить на 4 направления:

- коммуникации "пересечения памяти" (cross memory communications);
- явное использование дополнительных АП (AR ACS mode);
- пространства данных (data spaces);
- гиперпространства (hiperspaces).

Коммуникации пересечения памяти позволяют программе передавать управление в другое АП. Управление передается не "напрямую", а через системный вызов (блок запроса SRB). Различают синхронные и асинхронные коммуникации пересечения памяти.

В так называемом первичном режиме AR-программа работает только с данными, расположенными в первичном АП. В режиме же управления памятью через регистры доступа – режиме ACS AR-программа может определять регистры AR, используемые для трансляции адресов и, таким образом, употреблять обычные команды обращения к данным для работы с

параллельными АП. Программа, однако, не может передавать управление в другое АП, для этого режим ACS AR надо комбинировать с коммуникациями пересечения памяти.

Пространства данных и гиперпространства являются дополнительными именованными АП размером от 4 Кбайт до 2 Гбайт, используемыми только для размещения данных.

Программа, использующая пространства данных, должна работать в режиме ACS AR. Она использует системные вызовы для создания и удаления пространства данных и управления им, команды же, выполняемые в основном АП, могут непосредственно манипулировать данными в пространстве данных.

Программа, использующая гиперпространства, может работать в первичном режиме AR. Она использует системные вызовы для создания и удаления пространства данных и управления им, а также для того, чтобы пересылать данные между гиперпространством и основным АП. Обмен между первичным АП и гиперпространством ведется 4-Кбайтными блоками.

Пространства данных или гиперпространства могут содержать также и программные коды, но передавать управление на эти коды непосредственно программа не может. Для выполнения эти коды должны быть пересланы в буфер в первичном АП. Физически оба вида пространств могут размещаться как в основной, так и в расширенной памяти, но для пространства данных предпочтение отдается основной памяти, а для гиперпространства – расширенной. Для управления пространствами данных система использует те же механизмы страничного обмена, что и для первичного АП. Поскольку же манипулирование данными в гиперпространстве несколько ограничено, для управления гиперпространством используются более простые и более эффективные алгоритмы.

В z/OS имеется также механизм отображения в память объектов данных (data-in-virtual), аналогичный файлам, отображаемым в память в Unix. Этот механизм позволяет назначить "окно" виртуальных адресов, просматривать в этом окне нужную часть объекта данных и перемещать окно по мере необходимости. Отображение данных возможно (и предпочтительно) в пространство данных или в гиперпространство.

Приложение получает память в своем виртуальном АП, используя системные вызовы явного (GETMAIN или STORAGE) или неявного выделения памяти. Управление выделением памяти ведется при помощи так называемых подпулов. Подпулы состоят из 4-Кбайтных блоков памяти и формируются динамически: блоки добавляются в подпул или удаляются из него по мере необходимости. Система удовлетворяет каждый запрос на выделение памяти из одного блока (разумеется, кроме тех случаев, когда размер запроса превосходит размер блока). При размещении небольших запросов система ищет свободное место в уже выделенных блоках по принципу "самый подходящий" и лишь при невозможности удовлетворить запрос таким образом выделяет новый блок.

При создании любой задачи для нее обязательно создается подпул 0, но в задаче может быть создано и большее число подпулов. Любой подпул может использоваться только одной задачей или разделяться несколькими задачами. Подпул 0 разделяется задачей со всеми ее подзадачами, но если подпул 0 задачи определен как неразделяемый, для каждой подзадачи будет создаваться свой подпул 0. Монопольно используемые подпулы освобождаются с окончанием задачи, которая ими владеет. Разделяемые подпулы сохраняются, пока сохраняется хотя бы одна из использующих их задач.

"64-разрядная революция" аппаратуры мейнфреймов осваивается z/OS за несколько шагов.

Первый шаг, сделанный в z/OS V1R1, обеспечил 64-битное управление реальной памятью, что позволило уменьшить страничный обмен и ограничения на память для прежних 31-разрядных приложений.

Со второго шага, сделанного в z/OS V1R2, обеспечивается поддержка 64-битной адресации в одном АП. Качественно картина виртуальной памяти приложения остается такой же, как и представленная на рисунке 12.6, но выше границы 2 Гбайт, называемой "планкой" (bar) появляется дополнительная часть частного АП. Любая 31-битная программа может теперь получать виртуальную память за планкой и манипулировать данными в этой памяти. Программа по-прежнему размещается в пределах 2 Гбайт, виртуальная память выше планки предназначена только для данных. Новый язык Ассемблера включает в себя новые команды для работы с данными за планкой и манипулирования с 64-разрядными регистрами общего назначения. Системные вызовы для работы с данными за планкой включают в себя прежние механизмы выделения и освобождения памяти – для совместимости со старыми программами, но система управляет пространством выше планки как объектами данных. В новых механизмах программа создает за планкой объекты данных, размер которых кратен 1 Мбайту. В V1R2 объекты данных не могут совместно использоваться в разных АП.

Третий шаг (z/OS V1R3) состоит во внедрении AMODE=64. В сочетании с новыми возможностями Редактора Связей и Загрузчика этот режим позволяет создавать полностью 64-разрядные программы.

Четвертый шаг, который будет реализован в следующей версии z/OS, – обеспечение возможности разделения объектов данных, размещенных за планкой между разными АП.

Параллельно с введением 64-разрядных возможностей в системные сервисы и низкоуровневое программирование они внедряются и в

инструментальные средства (C/C++, Java), и в основные продукты промежуточного программного обеспечения (DB2, WebSphere и др.).

Управление процессами

АП в z/OS создается для задания. Как и в VSE, задание в z/OS состоит из нескольких последовательно выполняющихся программ – шагов задания. Для каждого шага задания создается задача (процесс). Структурой, представляющей задачу в системе, является блок управления задачей TCB (Task Control Block). Задача может порождать новые задачи (подзадачи) при помощи системных вызовов ATTACH (подзадача выполняется в первичном режиме AR) или ATTACHX (подзадача выполняется в режиме ASC AR). Порождаемые таким образом подзадачи имеют собственные блоки TCB и, таким образом, представляются как полноценные задачи, но все они выполняются в том же АП, что и породившая их задача. Порожденная подзадача выполняется параллельно с породившей и может порождать собственные подзадачи. Между задачей и ее подзадачами устанавливаются отношения "предок – потомок". Задача и ее подзадачи выполняются асинхронно, но выполнение задачи-предка может быть и синхронизировано с завершением задачи-потомка. Подзадача может порождаться с параметрами ЕСВ или/и EXTR. Первый параметр назначает для подзадачи Блок Управления Событием (Event Control Block), в котором делается отметка о завершении подзадачи. Если подзадача создается с параметром ЕСВ, ее TCB не удаляется с ее завершением, а сохраняется до тех пор, пока информация о завершении не будет востребована задачей-предком. Задача-предок может ожидать завершения потомка и получить информацию о его завершении, применяя системный вызов WAIT, параметром которого является ЕСВ потомка.

Параметр EXTR позволяет определить в задаче-предке процедуру, которая автоматически выполняется при завершении данного потомка.

Приоритеты выполнения определяются на двух уровнях:

- приоритет АП;
- приоритеты задач и подзадач.

Приоритет АП, как правило, определяется ОС из соображений обеспечения наилучшей загрузки системы (см. ниже). Однако этот приоритет может быть назначен и пользователем в операторах языка управления заданиями. Приоритет пользователя назначается дифференцированно для каждого шага задания. Приоритет АП, установленный пользователем для шага задания, во время выполнения шага изменяться в программе не может, но может меняться ОС.

Создавая задачу для шага задания, ОС присваивает ей диспетчерский (текущий) и граничный приоритеты. Подзадача по умолчанию наследует приоритеты своего родителя, но ей могут быть назначены и собственные приоритеты. Приоритеты подзадач могут меняться в программе системным вызовом SNAR. Приоритеты задач/подзадач, однако, никак не влияют на то, в каком порядке выбираются на выполнение программы – этот порядок определяется только приоритетом АП. Приоритеты же подзадач влияют на порядок их выборки на выполнение в пределах процессорного обслуживания, выделяемого для АП. В этих пределах приоритеты подзадач являются абсолютными: на выполнение выбирается подзадача с наивысшим приоритетом, текущая активная подзадача немедленно вытесняется, если подзадача с более высоким приоритетом приходит в состояние готовности.

Средства взаимодействия

Выше мы упомянули о блоках ЕСВ, используемых для синхронизации выполнения задачи и ее подзадач. Однако блоки ЕСВ

являются более универсальным средством синхронизации выполнения. ЕСВ используется с системными вызовами WAIT, POST и EVENTS. Системный вызов POST сигнализирует о совершении события, системные вызовы WAIT и EVENTS задерживают выполнение задачи до совершения одного или нескольких событий.

Для взаимного исключения доступа к ресурсам из разных программ используются системные вызовы ENQ и DEQ. В первом приближении их можно считать эквивалентным семафорным операциям P и V соответственно. При употреблении этих системных вызовов задается имя ресурса и масштаб. Имя (оно состоит из двух частей) в документации IBM называется именем ресурса, но на самом деле это скорее имя семафора, имена в операциях ENQ/DEQ назначаются произвольно и не имеют никакой связи с действительными именами ресурсов. Масштаб определяет область видимости ресурса:

- масштаб STEP означает, что ресурс доступен только в данном АП;
- масштаб SYSTEM означает, что ресурс доступен во всех АП, но только в данной вычислительной системе;
- масштаб SYSTEMS означает, что ресурс доступен во всех АП всех вычислительных систем sysplex'a.

Комбинация имени ресурса и масштаба должна быть уникальной.

В отличие от обычных семафорных операций, системный вызов ENQ может выполняться в монопольном или разделяемом режиме, что соответствует классической задаче "читатели–писатели". Любое число задач может одновременно получать доступ к ресурсу в разделяемом режиме, только одна задача может иметь доступ к ресурсу в монопольном режиме.

Системный вызов ENQ может также выполняться и с условием. Предусмотренные для ENQ условия могут обеспечивать:

- проверку состояния ресурса без его захвата;
- захват ресурса только в том случае, если он немедленно доступен;
- изменение режима захвата с разделяемого на монопольный;
- захват ресурса только в том случае, если программа еще им не владеет.

Задачи, задержанные при выполнении операции ENQ, образуют очереди к ресурсам, которые обслуживаются по дисциплине FCFS. Размер такой очереди может быть, однако, ограничен, в этом случае попытка выполнения запроса ENQ, который переполнит очередь, приводит не к блокированию программы, а к завершению запроса с признаком ошибки.

Попытка захвата ресурса, которым программа уже владеет, приводит к аварийному завершению программы. Ответственность за обход тупиков лежит на программисте.

Системные вызовы в z/OS выполняются системными сервисными процедурами. Такая процедура представляется в системе Блоком Сервисной Процедуры SRB (Service Routine Block). SRB во многом аналогичен TCB, но SRB не может владеть АП. Однако SRB может получать и использовать память в АП вызвавшей его задачи и в системном АП. После вызова процедуры и создания для нее SRB сервисная процедура может выполняться параллельно с вызвавшей ее программой (даже с реальным параллелизмом – на разных процессорах). Все системные процедуры являются реентерабельными и, следовательно, могут быть прерваны в ходе своего выполнения.

Подсистемы и управление ресурсами

Прежде чем рассмотреть принципы распределения ресурсов в системе, дадим краткие характеристики некоторым (далеко не всем) подсистемам в составе z/OS.

Подсистема ввода заданий JES (Job Entry Subsystem) обеспечивает выполнение пакетных заданий. Ее функции во многом аналогичны подсистеме POUWER в VSE. JES интерпретирует операторы языка управления заданиями JCL и управляет очередями. В системе могут быть запущены несколько копий JES, каждая из которых создает свой системный образ. Имеются две версии JES – JES2 и JES3, которые различаются тем, что в JES2 выполняется независимое управление каждой запущенной копией, в JES3 осуществляется централизованное управление всеми копиями.

Подсистема разделения времени TSO/E (Time Sharing Option/Extention) – основной интерактивный интерфейс z/OS. TSO/E обеспечивает для конечных пользователей, программистов и администраторов набор команд и полноэкранных возможностей для подготовки программ, подготовки и выполнения заданий, выполнения управления системой. Как обязательная часть z/OS, TSO/E является базой для ряда других системных сервисов, таких как Book Manager, Hardware Configuration Definition и другие.

z/OS UNIX System Services обеспечивают использование z/OS как сверхмощного Unix-сервера. Службы приложений z/OS UNIX System Services включают в себя командный интерпретатор shell, утилиты и отладчик. Набор команд и утилит полностью соответствует спецификациям стандарта Single Unix Specification, известного также как Unix 95. Это позволяет программистам и пользователям, даже не знающим команд z/OS, взаимодействовать с z/OS как с Unix-системой. Отладчик предоставляет программистам набор команд для интерактивной отладки программ, написанных на языке C. Этот набор подобен аналогичным

командам, существующим в большинстве Unix-систем. Службы ядра z/OS UNIX System Services совместно с языковыми средами обеспечивают соответствующий Single Unix Specification API для программирования на языке C, многопоточность и средства разработки клиент/серверных приложений. Это обеспечивает возможность программирования для z/OS как для Unix и переноса в z/OS приложений, созданных для Unix.

Еще MVS прошла сертификацию по стандартам POSIX, Single Unix Specification и OSF/1. Таким образом, z/OS соответствует Unix-ориентированным стандартам лучше, чем большинство систем, относящихся к семейству Unix, и является наилучшим Unix-суперсервером.

Планированием распределения ресурсов занимается Менеджер Системных Ресурсов SRM (System Resource Manager), являющийся компонентом BCP. SRM определяет, какие АП получают доступ к системным ресурсам, и ту долю системных ресурсов, которая будет выделена каждому АП. SRM распределяет ресурсы между АП в соответствии с приоритетными требованиями, заданными в параметрах инсталляции, и стремится достичь оптимального использования ресурсов с точки зрения производительности системы. При определении параметров функционирования SRM работы, выполняемые в системе, разбиваются на группы, называемые доменами. Домены характеризуются общими характеристиками работы, и общим для домена показателем важности. Каждое выполняющееся АП попадает в тот или иной домен – пакетное задание, транзакция IMS, транзакция DB2, короткая или длинная команда TSO и т.д. Управление доменами дает возможность:

- гарантировать доступ к системным ресурсам хотя бы минимальному количеству АП, принадлежащих к определенному типу работы;

- ограничивать количество АП, имеющих доступ к ресурсам, для каждого типа работ;
- назначать степень важности для каждого типа работ.

Управление характеристиками выполнения позволяет дифференцировать выполняемые работы, например, установить приоритет коротких транзакций над длинными, приоритет времени реакции над пропускной способностью и т.д.

Управление доменами позволяет установить, какие АП получают доступ к системным ресурсам. Диспетчеризация управляет долей системных ресурсов, получаемых каждым из допущенных АП. После того, как АП включено в мультипрограммный набор (набор АП, размещенных в основной памяти и допущенных к использованию ресурсов), все АП конкурируют за обладание ресурсами независимо от доменов, к которым они принадлежат. Диспетчеризация ведется по приоритетному принципу: работа с наивысшим приоритетом получает ресурс первой. Всего в системе имеется 256 уровней приоритетов, которые разбиты на 16 наборов по 16 уровней в каждом. Внутри каждого набора АП может иметь переменный или фиксированный приоритет. Фиксированные приоритеты более высокие, чем переменные. Фиксированный приоритет просто назначается АП в соответствии с параметрами, указанными в настройках SRM для домена. Переменные приоритеты периодически перевычисляются по алгоритму минимизации среднего времени ожидания.

SRM управляет использованием ресурсов в пределах всей системы и постоянно ищет пути преодоления дисбаланса – перегрузки ресурса или его простоя. Это достигается путем периодического пересмотра уровня мультипрограммирования – количества АП, которые находятся в основной памяти и готовы к диспетчеризации. Когда характеристики использования показывают, что ресурсы системы используются не полностью, SRM выбирает домен и увеличивает число допущенных АП в этом домене. Если показатели использования говорят о перегрузке системы, SRM уменьшает уровень мультипрограммирования.

Для уменьшения интенсивности страничного обмена SRM применяет так называемый "логический свопинг". Страничные фреймы АП,

подвергающегося логическому свопингу, сохраняются в основной памяти на время, не превышающее некоторого порогового значения, устанавливаемого SRM. Пороговое значение для логического свопинга перевычисляется динамически и зависит от текущей потребности системы в основной памяти.

SRM автоматически определяет наилучший состав АП в мультипрограммном наборе и количество основной памяти, выделяемое каждому АП, наиболее эффективное в рамках принятого уровня мультипрограммирования. При этом управление страничным обменом и использованием ЦП в рамках всей системы сочетается с индивидуальным управлением рабочим набором каждого АП. Таким образом, показатели свопинга определяются общесистемными показателями страничного обмена и требованиями рабочего набора, причем последние имеют некоторый приоритет.

SRM также определяет приоритеты АП в очередях ввода-вывода. По умолчанию эти приоритеты такие же, как и диспетчерские приоритеты, но в параметрах SRM для доменов могут быть назначены приоритеты ввода-вывода выше или ниже их диспетчерских приоритетов.

SRM управляет распределением дисковых устройств и контролирует использование таких ресурсов, как вторичная память (дисковые области, используемые для свопинга – они не входят в дисковое пространство, управляемое файловой системой), область системных очередей и ресурс страничных фреймов. При нехватке этих ресурсов SRM предпринимает меры, сводящиеся к уменьшению уровня мультипрограммирования.

Настройка SRM производится при инсталляции ОС и продолжается в ходе ее эксплуатации. Это процесс итеративный и, возможно, бесконечный, так как в ходе эксплуатации характеристики выполняемого системой потока работ могут уточняться и меняться. Мы уже отмечали в нашей книге, что мейнфреймы обладают весьма высоким показателем производительность/стоимость, но реально высоким этот показатель может быть только тогда, когда производительность будет востребована в полном объеме. Эффективность работы SRM существенно зависит от параметров, заданных при его настройки, а гарантировать правильность определения пользователем большого числа параметров, многие из которых могут находиться в сложной зависимости друг от друга, невозможно. Поэтому

возникла необходимость переложить планирование нагрузки в вычислительной системе или sysplex'е на ОС. В настоящее время надстройка над SRM, осуществляющая это планирование – Менеджер Нагрузки (WLM – Workload Manager) – включена в ядро ОС. WLM требует от администратора нагрузки задания определения сервиса и сам реализует это определение в рамках системы или sysplex'а. Определение сервиса производится не в терминах системных параметров, как для SRM, а в терминах пользователя. Таким образом, WLM требует от пользователя определение того, что нужно сделать, а не того, как это делать. Как это делать, WLM решает сам, учитывая конфигурацию системы и требования всех используемых подсистем, обеспечивающих собственные среды выполнения – как входящих в состав ОС (TSO, JES, Unix System Services и др.), так и отдельных программных продуктов (CICS, DB2, MQSeries и др.).

Определение сервиса включает в себя:

- Политику сервиса – набор бизнес-целей управления, таких как время реакции и максимальная задержка выполнения. Каждой цели придается также весовой коэффициент, характеризующий важность достижения этой цели.
- Классы сервиса, которые разбиваются на "периоды" – группы работ с одинаковыми целями и требованиями к ресурсам.
- Группы ресурсов, которые определяют границы процессорной мощности в sysplex'е. Назначая классу сервиса группу ресурсов, администратор загрузки определяет минимальный и максимальный объем процессорного обслуживания для работы.
- Правила классификации, которые определяют, как отнести поступившую работу к тому или иному классу сервиса.
- Прикладные среды – группы прикладных функций, которые выполняются в АП сервера и могут быть вызваны клиентом. WLM в соответствии с определенными целями автоматически активизирует или останавливает АП сервера.

- Среды планирования – списки ресурсов (первичных и вторичных) с отражением их состояния, которые позволяют гарантировать, что система (в составе sysplex'a) обладает достаточным ресурсом для выполнения работы.
- Классы отчетности, используемые для получения более подробной информации о производительности внутри класса сервиса.

В соответствии с определением сервиса WLM обеспечивает распределение нагрузки между процессорами одной вычислительной системы и системами всего sysplex'a, управление временем задержки работы после прихода ее в состояние готовности, управление анклавами (транзакциями, например, DB2, выполняющимися параллельно в разных АП, возможно, на разных системах sysplex'a), управление запросами клиентов к серверами и получение информации о состоянии. Управление ведется WLM с динамической обратной связью, с учетом нагрузки в каждый текущий момент, а также распределения нагрузки на предыдущем интервале времени.

Следующим шагом в оптимизации использования ресурсов мейнфреймов и их sysplex'ов стало внедрение Интеллектуального Распорядителя Ресурсами IRD (Intellegent Resource Director). IDR обеспечивает возможность управлять несколькими образами операционной системы, выполняющимися на одном сервере (в разных логических разделах), как одним вычислительным ресурсом с динамическим управлением нагрузкой и балансировкой физических ресурсов – процессоров и каналов ввода-вывода – между многими виртуальными серверами. Система динамически перераспределяет эти ресурсы в соответствии с определенными бизнес-приоритетами с тем, чтобы удовлетворить непредсказуемые требования задач электронного бизнеса. IRD включает в себя три основных компонента:

- LPAR CPU Management – логические разделы (LPAR) на одном z-сервере объединяются в виртуальные sysplex-кластеры и управляются в соответствии с бизнес-целями и их важностью, сформулированными для WLM;
- Dynamic Channel Path Management – дает возможность динамически переключать каналные пути (через переключатель ESCON Director) от одного контроллера к другому, таким образом, WLM получает возможность обеспечивать раздел большей или меньшей пропускной способностью по вводу-выводу – в соответствии с требованиями и важностью выполняемой в нем задачи;
- Channel Subsystem Priority Queuing – распространяет возможности приоритетной диспетчеризации ввода-вывода на весь LPAR-кластер: если важная задача не может обеспечить выполнение своих целей из-за нехватки пропускной способности ввода-вывода, раздел, выполняющий эту задачу, получает дополнительные каналы ввода-вывода.

IRD является частью широкомасштабного проекта IBM eLiza, целью которого является создания фундамента для информационных систем с уменьшенной сложностью и стоимостью эксплуатации, использования, администрирования. Хотя проект eLiza не ориентирован на единственную аппаратную платформу и операционную среду, по вполне понятным причинам z-серверы и z/OS являются "передним краем" его реализации. Цели проекта eLiza сформулированы как: самооптимизация (self-optimization), самоконфигурирование (self-configuration), самовосстановление (self-healing) и самозащита (self-protection).

Самооптимизация заключается в свойствах WLM и IRD эффективно перераспределять ресурсы в условиях непредсказуемой рабочей нагрузки.

В z/OS V2 планируется распространить возможности IRD на разделы, выполняющие ОС, отличные от z/OS (z/VM, Linux).

Самоконфигурирование поддерживается такими средствами, как msys for Setup (обеспечение простоты для пользователя установки программного обеспечения) и z/OS Wizards – web-базируемые диалоговые средства настройки системы.

Самовосстановление поддерживается:

- множеством функций контроля и восстановления оборудования (Hardware RAS), среди которых – определение различных сбоев и в ряде случаев – автоматическое переключение и восстановление, plug and play и "горячее" переключение ввода-вывода и др.;
- дуплексной передачей структур соединения (System-Managed CF Structure Duplexing) – устойчивым механизмом, позволяющим обеспечить неразрывное соединение;
- msys for Operations – обеспечением лучшей работоспособности приложений за счет большей информации о ресурсах и самовосстановления критических ресурсов, а также снижения вероятности и облегчения восстановления из-за ошибок оператора;
- System Automation for OS/390 – продуктом, обеспечивающим восстановление приложений, ресурсов системы и sysplex'a на базе принятой политики обслуживания.

Самозащита обеспечивается:

- Intrusion Detection Services – средством, позволяющим обнаруживать атаки на систему и выбирать механизмы защиты;
- Public Key Infrastructure – встроенной в z/OS системой аутентификации и авторизации на основе открытого ключа;
- средствами безопасности, являющимися промышленными стандартами: LDAP, Kerberos, SSL, цифровые сертификаты и т.д.

Часть описанных средств уже имеется в составе z/OS, в рамках проекта eLiza предполагается их интеграция, расширение их возможностей в новых версиях, создание новых средств и перенос их на другие аппаратные платформы и в другие ОС IBM.

12.4. Операционная система z/VM

ОС z/VM [21, 24, 42] (последняя версия – V4R2) является высокопроизводительной многопользовательской интерактивной ОС, предоставляющей уникальные возможности в части выполнения различных операционных сред на одном вычислительном комплексе, поддержки интерактивных пользователей и клиент/серверных сред. Существует "легенда" о том, что VM родилась как инструментальное средство, предназначенное для использования только внутри IBM, и попала на рынок вопреки планам фирмы. Хотя IBM и опровергает эту легенду, она выглядит вполне правдоподобно. z/VM представляет интерес для применения прежде всего в таких случаях:

- как инструментальная платформа для разработки и отладки системного программного обеспечения, в том числе и других ОС, обеспечивающая эффективное использование вычислительных мощностей в процессе отладки и легкое восстановление после краха отлаживаемой системы;
- как платформа для миграции на новые ОС и новые версии ОС и системного программного обеспечения, обеспечивающая параллельное функционирование как старого, так и нового программного обеспечения;
- как платформа для информационных систем, требующих параллельного функционирования множества прикладных и операционных сред, хорошо друг от друга изолированных, с одной стороны, а с другой, легко устанавливающих связи друг с другом.

Уникальные свойства z/VM определяются ее архитектурой. Аббревиатура VM расшифровывается как Virtual Machine, и эта ОС в полной мере воплощает концепцию виртуальных машин: интерфейс процесса выглядит как интерфейс оборудования. Ядро z/VM составляет Управляющая Программа CP (Control Program), которая предоставляет для своих конечных пользователей рабочую среду, называемую виртуальной машиной (VM). VM в z/VM является аналогом процесса в других ОС: это тот "субъект", которому CP выделяет ресурсы. VM моделирует реальную вычислительную систему: процессор (или процессоры), память, устройства и каналы ввода-вывода. У пользователя создается впечатление, что в его распоряжении имеется реальная ЭВМ, доступная для него в привилегированном режиме. На самом же деле, в его распоряжении находится только то подмножество ресурсов, которое выделяет или моделирует для VM CP. PSW VM определяет для выполняющейся на VM программы состояние "супервизор" (привилегированное состояние). PSW же реального оборудования при выполнении такой программы определяет состояние "задача" (непривилегированное). При попытке программы, выполняющейся на VM, выполнить привилегированную команду происходит исключение, и управление получает CP. CP распознает причину исключения и выполняет для VM привилегированную команду или моделирует ее выполнение, после чего возвращает управление VM. Исключение и его обработка скрыты от VM, VM кажется, что ее привилегированная команда выполнилась на реальном оборудовании.

CP на выбор моделирует для VM архитектуры нескольких поколений мейнфреймов – от 370/XA до z900, а также виртуальную архитектуру ESA/XC (eXtended Configuration), в которой VM могут быть доступны (при авторизации) адресные пространства других VM. В число компонентов архитектуры VM входят:

- процессор/процессоры;

- память;
- внешняя память;
- операторская консоль;
- каналы и устройства ввода-вывода.

Поскольку CP предоставляет ВМ модель, неотличимую для нее от ресурсов реальной вычислительной системы, программа, выполняющаяся на ВМ, может (и должна) осуществлять управление этими ресурсами, то есть в свою очередь быть операционной системой. Такие ОС называются в z/VM гостевыми (guest). В документации z/VM CP иногда называют гипервизором, в отличие от супервизоров – управляющих программ гостевых ОС. Гостевая ОС может "знать" о том, что она работает под управлением гипервизора, в этом случае гостевая ОС может использовать обращения к CP (команда DIAGNOSE), а также гостевая ОС и CP могут распределять между собой управление ресурсами: гипервизор работает с интерфейсом оборудования, а гостевая ОС – с интерфейсом процесса. Если же гостевая ОС не знает о присутствии CP, то она выполняет управление созданной для нее моделью ресурсов в полном объеме. С этой точки зрения можно разделить гостевые ОС на четыре категории:

1. ОС, специально созданные как гостевые, которые могут работать только в среде ВМ под управлением гипервизора – CMS и GCS.
2. Полнофункциональные другие ОС мейнфреймов (VSE, z/OS и ее предшественники, Linux for zSeries), выполняющиеся "не зная" о существовании гипервизора.
3. Те же ОС, но адаптированные для выполнения в среде ВМ, адаптация состоит в том, что исключается дублирование функций в гипервизоре и супервизоре.
4. Гостевой ОС может быть другая (вторичная) CP, которая распределяет выделенное ей подмножество ресурсов между

своими ВМ и своими гостевыми ОС, среди которых в свою очередь могут быть СР и т.д.

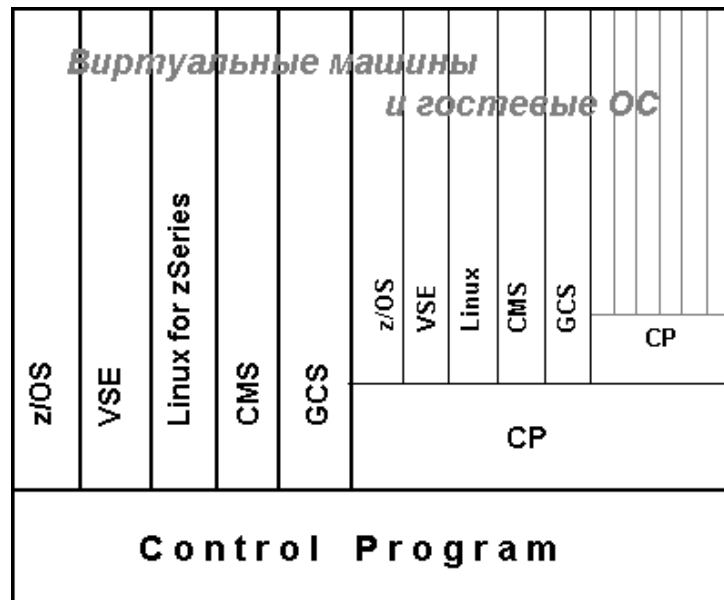


Рисунок 12.7 СР и виртуальные машины

Определение ВМ является квазипостоянным: оно создается один раз, а затем используется многократно. Определение ВМ сохраняется в каталоге СР, основным содержанием элемента каталога СР является описание ресурсов, выделяемых ВМ. При запуске ВМ на выполнение СР на основе элемента каталога строит Блок Определения Виртуальной Машины – VMDBLK (Virtual Machine Definition BLock), в котором содержится описание ресурсов ВМ (либо непосредственно в VMDBLK, либо как ссылки на другие управляющие блоки) и их текущего состояния. Если для ВМ создается несколько виртуальных процессоров, то для каждого процессора создается свой VMDBLK, но только один из VMDBLK каждой ВМ является базовым – тот, который содержит описание памяти ВМ. Свой VMDBLK имеет также и СР. Все VMDBLK связаны в кольцевой список.

Управление памятью

Возможно, главным ресурсом, которым управляет СР, является реальная память, и с этой точки зрения СР может создавать ВМ трех типов:

1. Тип $V=V$ – ВМ, которой выделяется только виртуальная память, требуемый размер памяти для ВМ обеспечивается за счет динамической трансляции адресов и страничного свопинга.
2. Тип $V=F$ – ВМ, которой выделяется непрерывная область реальной памяти. Эта область исключается из страничного обмена, но динамическая трансляция адресов для ВМ $V=F$ применяется, так как виртуальное адресное пространство ВМ начинается с адреса 0, а в реальной памяти область, выделяемая для ВМ $V=F$, начинается не с 0. ВМ типа $V=F$ обладают преимуществом в производительности перед ВМ $V=V$.
3. Тип $V=R$ – ВМ, которой выделяется непрерывная область реальной памяти, начиная с адреса 0. Эта память исключается из страничного обмена и для нее не применяется динамическая трансляция адресов. Кроме того, ВМ $V=R$ может выполнять некоторые привилегированные операции на реальном оборудовании. Очевидно, что производительность ВМ этого типа наивысшая.

ВМ двух последних типов называются привилегированными. В настоящее время СР допускает одновременное функционирование не более шести привилегированных ВМ, из которых только одна может быть типа $V=R$, тогда как число одновременно работающих ВМ типа $V=V$ может исчисляться десятками тысяч. Работа привилегированных ВМ резко отрицательно сказывается на производительности всех других ВМ, поэтому эти типы ВМ создаются только при наличии действительной необходимости в них (например, для задач реального времени).

Если под управлением СР работают только ВМ типа $V=V$, то ядро СР занимает нижнюю часть реальной памяти (начиная с адреса 0). Вся реальная память выше ядра отводится под динамическую страничную область, которая подвергается страничному обмену. Если же под управлением СР работают наряду с ВМ типа $V=V$ и привилегированные ВМ, то нижняя часть реальной памяти отводится под область $V=R$. Часть этой области, начиная с адреса 0, занимает единственная ВМ типа $V=R$, остальная часть области распределяется между ВМ типа $V=F$. Выше области $V=R$ размещается ядро СР, а еще выше – динамическая страничная область.

Динамическая страничная область содержит:

- управляющие блоки СР;
- нерезидентные модули СР;
- блоки управления памятью СР;
- буферы для спулинга и файловой системы;
- префиксные страниц для реальных процессоров;
- свободные страничные кадры;
- страницы ВМ типа $V=V$.

В архитектуре z/VM различаются три уровня памяти:

- память первого уровня – реальная память;
- для каждой ВМ СР строит виртуальное адресное пространство – память второго уровня;
- с точки зрения гостевой ОС память второго уровня является реальной памятью, и гостевая ОС строит для своих процессов виртуальные адресные пространства – память третьего уровня.

Виртуальная память ВМ состоит из сегментов. Для каждой ВМ строится своя таблица сегментов. При размере виртуальной памяти ВМ до 32 Мбайт таблица сегментов находится непосредственно в VDMBLK, при

большем размере – для нее выделяются дополнительные страницы (по 1 странице на каждые 1024 Мбайт виртуальной памяти). С каждой таблицей сегментов связана собственная таблица страниц.

Страницы, размещаемые в динамической страничной области, подвергаются вытеснению и подкачке. СР ведет общий список свободных страничных кадров и списки используемых страничных кадров для каждой ВМ.

Список свободных страничных кадров пополняется при необходимости и обрабатывается по дисциплине LIFO.

Списки используемых страничных кадров ВМ обрабатываются по дисциплине рабочего набора. Этот список периодически переупорядчивается по частоте использования. После каждого такого переупорядочивания в списке устанавливается промежуточный указатель – на начало той части списка, в которой располагаются дескрипторы кадров, к которым не было обращения со времени последнего переупорядочивания. Первая часть списка составляет рабочий набор ВМ, вторая – страницы-кандидаты на перенос в список свободных страничных кадров.

Если размер списка свободных страничных кадров достигает установленной нижней границы, происходит пополнение списка. Список пополняется до тех пор, пока его размер не достигнет верхней установленной границы. Это может потребовать неоднократного просмотра списков страничных кадров ВМ, каждый следующий просмотр предъявляет более жесткие требования к состоянию страницы и ВМ, которой страничный кадр принадлежит.

При выделении для ВМ виртуальной памяти предусмотрены два механизма: выделение блока памяти произвольного размера и выделение блока из подпула. Выделение блока произвольного размера происходит традиционными методами. Выделение из подпула выполняется быстрее произвольного и применяется для выделения памяти (как правило,

неявного) для управляющих блоков, создаваемых для ВМ. В этом случае выделяется один из заранее заготовленных блоков стандартного размера.

Диспетчеризация ВМ

При работе на двух- и более процессорной конфигурации реальной системы для ВМ типа $V=R$ по умолчанию выделяется отдельный процессор. Для ВМ типа $V=F$ отдельный процессор может быть выделен, но по умолчанию это не делается.

СР может моделировать многопроцессорную конфигурацию для ВМ, но при создании ВМ с большим числом процессоров, чем имеется в реальной системе, производительность такой ВМ падает, поэтому такой прием применяется только при отладке гостевых ОС и другого программного обеспечения, рассчитанного на многопроцессорную конфигурацию.

Единицей диспетчеризации с точки зрения распределения процессорного обслуживания является ВМ. Основной целью обслуживания является справедливое распределение процессорного времени между ВМ. СР поддерживает три очереди ВМ на процессорное обслуживание:

- диспетчерскую очередь, d-список (dispatch list), ВМ состоящие в диспетчерской очереди, получают процессор в режиме разделения времени, мы называли такие процессы готовыми;
- очередь готовых ВМ, e-список (eligible list), которые исключены из диспетчеризации из-за нехватки ресурса памяти;
- список "спящих" ВМ (dormant list).

Готовыми здесь называются те ВМ, которые требуют выполнения какой-либо процессорной транзакции. "Спящие" ВМ процессорного обслуживания не требуют.

Все ВМ распределяются по четырем классам обслуживания:

- критические – те, для которых гарантируется отсутствие ожидания в e-списке (класс 0);
- очень интерактивные – выполняющие короткие транзакции (класс 1);
- интерактивные – выполняющие транзакции средней длительности (класс 2);
- неинтерактивные – выполняющие длинные транзакции (класс 3).

Класс с меньшим номером имеет более высокий приоритет в e-списке. В d-списке приоритеты перевычисляются динамически через каждый квант времени. При перевычислении кванта принимается во внимание:

- внешний приоритет ВМ;
- время ожидания в d-списке;
- интерактивная добавка для класса 1;
- страничная добавка – единоразовая добавка к приоритету, назначаемая для ВМ класса 2 или 3 при задержке из-за страничного отказа.

Очередная ВМ из d-списка получает квант процессорного времени, который назначается таким образом, чтобы его время было примерно равно времени выполнения 100000 машинных команд. Если ВМ переходит в состояние ожидания до исчерпания кванта, то она еще остается в d-списке на время, называемое "интервалом проверки ожидания" (300 мсек). Если ВМ выйдет из ожидания до истечения этого интервала, она остается в d-списке и получает возможность использовать недоиспользованный квант. Если время ожидания превосходит интервал проверки, ВМ переводится в список спящих. ВМ за время пребывания в d-списке разрешается трижды воспользоваться интервалом проверки ожидания.

Кроме того, СР также вычисляет для каждой ВМ квант готовности – общее реальное время пребывания ВМ в d-списке. Для ВМ класса 1 этот квант вычисляется системой таким образом, чтобы за время кванта готовности успели завершить свои транзакции 85% ВМ класса 1. Для классов 0 и 2 этот квант в 6 раз больше, чем для класса 1, для класса 3 – в 48 раз больше, чем для класса 1. Если ВМ исчерпала квант готовности, но не завершила транзакцию, она переводится в следующий класс.

Использование рабочего набора не влияет на приоритет ВМ в e-списке, но влияет на перемещение ВМ между d- и e-списками. Если ВМ, находящейся в d-списке, не хватает памяти для размещения своего рабочего набора, в d-списке блокируются все ВМ того же и большего класса. Если ВМ, находящаяся в d-списке превысила лимит роста своего рабочего набора (кроме ВМ класса 0), она переводится в e-список. Если в e-списке имеется ВМ класса 1, которая запаздывает с переходом в d-список, СР пытается вытеснить из d-списка последние переведенные в него ВМ классов 2 или 3.

В результате такой политики распределения процессорного обслуживания преимущество получают интерактивные ВМ, выполняющие короткие процессорные транзакции.

Виртуальные устройства

ВМ владеет также виртуальными каналами и устройствами. Назначение ВМ виртуального внешнего устройства может быть как постоянным (записанным в соответствующем данной ВМ элементе каталога СР), так и временным. С точки зрения ВМ виртуальное устройство ничем не отличается от реального, оно имеет в ВМ свой физический адрес и ВМ управляет им как реальным устройством.

Некоторые внешние устройства (например, накопители на магнитных лентах) закрепляются за ВМ. Закрепление означает, что

устройство используется ВМ в монопольном режиме, и управляющие воздействия, формируемые ВМ для устройства, почти не преобразуются СР. Однако, и в случае закрепления устройство для ВМ является виртуальным. Его адрес в ВМ не совпадает с реальным адресом устройства, СР преобразует адрес устройства в реальный, а также выполняет трансляцию адресов памяти в канальных программах, так как ВМ формирует канальную программу с адресами в своем АП. Как правило, устройства не закрепляются за ВМ постоянно, закрепление происходит при необходимости и отменяется при окончании работы с устройством.

z/VM также широко использует концепцию спулинга. Каждая ВМ имеет свой виртуальный принтер и виртуальные устройства ввода с перфокарт и вывода на перфокарты. Физически эти устройства моделируются очередями на внешней памяти. Если очередь принтера может быть выведена на реальное устройство, то данные из очередей перфокарточных устройств так и остаются на внешней памяти, так как реальные перфокарточные устройства просто уже не существуют. Но эти данные могут пересылаться из выходных очередей в выходные. Механизмы спулинга используются также для организации так называемых именованных сегментов памяти (named storage segment). В таких сегментах в области спулинга сохраняются многократно используемые коды и данные, например, образы гостевых ОС.

Каждое реальное дисковое устройство разделяется на несколько областей. Среди таких областей – область системных данных, вторичная память страничного обмена, область спулинга и области минидисков – постоянных и временных. Для обеспечения ВМ внешней памятью СР использует разделение дискового пространства. Каждая ВМ получает в свое распоряжение несколько минидисков. С точки зрения ВМ минидиск выглядит как реальное дисковое устройство с собственным физическим

адресом. На самом же деле минидиск – это лишь часть дисковой памяти, выделенная для ВМ в области минидисков на реальном дисковом накопителе. Описание минидисков, принадлежащих ВМ (адрес на реальном диске и размер), хранится в каталоге СР. Минидиски могут быть доступными только для чтения или для чтения/записи и использоваться в монопольном или совместном режиме. Обычно каждой ВМ назначается один или несколько минидисков для монопольного использования в режиме чтения/записи, а также разрешается доступ к нескольким минидискам, совместно используемым в режиме чтения (содержащим, например, системные утилиты, средства разработки и т.п.). Наряду с этим, ВМ может получать доступ к минидискам других ВМ – при соответствующей авторизации. Если минидиск разделяется двумя или более ВМ в режиме чтения/записи, то требуются специальные средства для предотвращения конфликтов и потери данных. Наряду с постоянно назначенными для ВМ минидисками, для ВМ могут создаваться и временные минидиски (создаваемые в области временных минидисков на реальном накопителе), и временные виртуальные минидиски (моделируемые буферами в памяти).

Примером еще одного подхода к виртуализации устройств в z/VM является виртуальный адаптер канал-канал. Через этот адаптер может осуществляться взаимодействие между ВМ. Управляющие воздействия, которые ВМ формирует для виртуального адаптера – такие же, как и для реального адаптера. Однако виртуальный адаптер не отображается ни на какое реальное устройство, он моделируется СР, а управляющие воздействия для него интерпретируются СР с использованием буферов в памяти и программных кодов.

CMS

Диалоговая управляющая система CMS (Conversation Monitor System) является гостевой ОС, обязательным компонентом z/VM. Это интерактивная однопользовательская, однозадачная ОС, предназначенная прежде всего для разработчиков программного обеспечения и администраторов системы. CMS разрабатывалась именно как гостевая ОС, поэтому ей не свойственны некоторые функции, типичные для самостоятельных ОС, такие как диспетчеризация и планирование, управление реальной памятью – эти функции выполняет СР. CMS обеспечивает работу с файловой системой, управление виртуальной памятью, управление виртуальными устройствами и интерфейс пользователя.

CMS не обеспечивает многозадачности. В программах, разрабатываемых для CMS, возможна многопоточность, но параллельная обработка обеспечивается только на уровне нитей, но не программ. Структура виртуальной памяти в CMS также очень проста. Нижнюю часть виртуального АП занимают структуры ядра CMS, создаваемые для каждой ВМ. Выше расположено частное адресное пространство программы, выполняющейся в CMS. Верхнюю часть АП занимают объекты, совместно используемые в режиме чтения: общая для всех ВМ часть структур и кодов CMS, система подсказки и т.п.

CMS состоит из следующих основных частей:

- терминальная система, обеспечивающая коммуникации между пользователем и ОС;
- системные службы CMS, в том числе:
 - команды и утилиты и пакетная служба;
 - сервис библиотек LIBRARYAN;
 - сервис редактора XEDIT;
 - командные интерпретаторы EXEC2, CMS EXEC, REXX;
 - Open Extention;

- файловые системы:
 - файловая система минидисков;
 - SFS;
 - BFS.

CMS является интерактивной командно-управляемой ОС и обладает богатым набором команд и утилит, обеспечивающих управление файлами, выполнение программ и управление системой. Хотя основной интерфейс CMS – командная строка, использование в утилитах возможностей REXX и XEDIT обеспечивает во многих случаях полноэкранный интерфейс взаимодействия с системой. Пользователи CMS могут также готовить пакетные задания, для этого в их распоряжении есть специальный командный язык. Пакетные задания пересылаются серверу пакетной обработки CMSBATCH, выполняющемуся в отдельной виртуальной машине и обслуживающему пакетных клиентов на всех ВМ системы.

Сервис библиотек обеспечивает создание и ведение библиотек макроопределений, объектных модулей и загрузочных модулей, а также загрузку и выполнение модулей из библиотек z/OS.

XEDIT является полноэкранным текстовым редактором с богатыми возможностями манипулирования текстом и форматирования экрана. Для XEDIT с помощью языка REXX могут создаваться сколь угодно сложные макрокоманды и профили, что позволяет использовать его как основу для создания интерфейсов системных и пользовательских утилит.

Наиболее развитым процедурным языком CMS является язык REXX. Этот язык родился именно в CMS, но сейчас является обязательной составной частью любой операционной системы IBM. В качестве прототипа для REXX был взят язык программирования PL/1, таким образом, REXX обладает полным набором алгоритмических возможностей и возможностью выполнять в программе команды, адресуемые операционной системе (CMS или CP) или другой системной или

прикладной среде (например, XEDIT, DB2 и т.д.). В отличие от своего прототипа, REXX является интерпретирующим языком и в полной мере использует это свойство – вплоть до возможности выполнить переменную – строку символов как оператор программы.

Файловые системы CMS

На минидисках, предоставляемых, ВМ CMS организует файловую систему с плоским каталогом, распределением пространства блоками по 512 байт и планом размещения файлов в виде B^+ -дерева. Минидиски идентифицируются буквами от A до Z, полное имя файла состоит из собственно имени, типа (аналог расширения в MS DOS, Windows или OS/2) и идентификатора диска. Файлы CMS – записеориентированные с постоянным или переменным размером записи.

Файловая система на минидисках была первой файловой системой для CMS, однако ее существенный недостаток состоит в том, что дисковое пространство для минидисков ВМ должно выделяться все сразу и, таким образом, реальное дисковое пространство используется нерационально. Поэтому для CMS была разработана Разделяемая Файловая Система SFS.

SFS обеспечивает совместное управление дисковым пространством для всех ВМ и динамическое выделение внешней памяти для ВМ в пределах установленной для нее квоты. Управление обеспечивается сервером SFS, выполняющимся на отдельной ВМ и обслуживающим все остальные ВМ в системе. Для каждой ВМ сервер SFS обеспечивает собственную структуру хранения файлов в виде дерева каталогов глубиной до 8 уровней. Корнем дерева является имя файлового пула и имя ВМ. Пользователь ВМ может давать права доступа к своим файлам и каталогам другим пользователям, в том числе, и право PUBLIC. SFS обеспечивает также алиасы и автоматическую защиту совместно используемых файлов

от одновременной записи. Специальные команды CMS обеспечивают возможность представления каталога SFS как минидиска или наоборот – минидиска как каталога SFS.

Управляющие и пользовательские данные SFS располагаются на минидисках сервера SFS и составляют файловый пул. Сервер обслуживает только один файловый пул, но в системе может быть запущено в нескольких ВМ несколько серверов SFS. Один минидиск сервера является управляющим, на нем находятся карты распределения памяти (память в SFS распределяется блоками по 4 Кбайт) и карты свободных блоков. Один или несколько минидисков отводятся под каталог, в котором хранится информация о пользователях, файлах, каталогах, алиасах и правах доступа. Минидиски каталога составляют так называемую группу памяти 1. Два минидиска отводятся под журнал транзакций, который ведет SFS для сохранения целостности своих управляющих данных. Эти два минидиска назначаются на разных реальных дисках и на разных контроллерах и хранят две идентичные копии журнала.

Остальные минидиски сервера составляют пользовательские данные, которые для удобства управления разбиваются на группы памяти. Всего возможно до 32К групп памяти, группы могут добавляться к файловому пулу. Размер всех групп памяти одинаков, группа состоит из нескольких минидисков, желательно – находящихся на разных реальных дисках.

CMS Open Extension

Чрезвычайно важным компонентом CMS является Open Extension, позволяющий CMS функционировать как Unix-системе. Open Extension обеспечивает выполнение ряда спецификаций стандартов PIOSIX, Single Unix Specification и DCE, как в части интерпретатора shell и утилит, так и в части API и файловой системы.. Для соблюдения стандарта POSIX в иерархическую файловую систему в SFS добавлено расширение,

называемое Байтовой Файловой Системой BSF (Byte File System). BFS в отличие от SFS обеспечивает байориентированное представление файлов, иерархическую структуру каталогов без ограничений на глубину вложенности, связи и символьные связи, права доступа к файлам. Open Extension позволяет разрабатывать в CMS POSIX-совместимые приложения и портировать таковые в CMS, а также функционировать выполняемым в CMS приложениям в гетерогенной распределенной среде.

GCS

Групповая Управляющая Система GCS (Group Control System), как и CMS является гостевой ОС – компонентом z/VM. GCS не конкурирует с CMS, они предназначены для разных задач. Если CMS – система для поддержки интерактивной работы, разработки и администрирования, то GSC – среда для выполнения приложений, прежде всего – приложений, тесно взаимодействующих друг с другом и приложений в архитектуре IBM SNA (System Network Architecture).

GSC позволяет объединять ВМ в группы, управляемые общим супервизором. Все ВМ в группе используют общий загруженный код ОС и ряда системных сервисов, а также имеют общую область памяти, доступную для чтения и записи.

Специфической функцией GCS в составе z/VM является поддержка архитектуры SNA как части z/VM без помощи какой-либо другой ОС. Эта поддержка выполняется продуктом ACF/VTAM (Advanced Communication Function/Virtual Telecommunications Access Method). Версия VTAM для GCS выполняется на одной из ВМ группы и управляет потоками данных, проходящими между сетевыми устройствами и программами, выполняющимися на других ВМ группы. VTAM также предоставляет сетевой интерфейс другим программным продуктам, обеспечивающим коммуникации, таким как:

- APPC (Advanced Program-to-Program Communications)/VM Support, обеспечивающий высокоуровневый интерфейс взаимодействия программ по протоколу APPC, независимый от того, являются взаимодействующие программы локальными или удаленными;
- RSCS (Remote Spooling Communications Subsystem), обеспечивающая передачу информации через сеть SNA, работу с файлами спулинга и передачу сообщений через связи не-SNA;
- NetView – средство управления сетями SNA.

Виртуальное АП, создаваемое GCS для выполняющихся в ней приложений отчасти напоминает АП приложений CMS: 16-Мбайтное пространство распределяется следующим образом (от меньших адресов к большему):

- управляющие блоки GCS, создаваемые для каждой ВМ;
- частное АП приложений;
- коды ядра общие управляющие блоки GCS (совместно используемые группой);
- общая область данных, общие управляющие блоки GCS (только чтение);
- общая область памяти (чтение и запись).

При расширении АП выше 16 Мбайт в верхней части АП создаются дополнительные порции частного АП и области данных и памяти.

В отличие от CMS, GCS является многозадачной ОС, поэтому задача управления памятью для нее сложнее: нужно обеспечить динамическое выделение памяти для нескольких программ и защиту памяти одной программы от другой. Для разделения областей памяти, принадлежащих разным программам, GCS использует механизм ключей защиты памяти (описанный в разделе 12.1). При запросе программы на выделение памяти GCS ищет свободную страницу, ключ защиты которой совпадает с ключом

программы, выдавшей запрос, при отсутствии таковой – изменяет ключ защиты любой другой свободной страницы.

GSC поддерживает многозадачность на одной VM. Следует, однако, помнить о том, что распределение процессорного обслуживания между программами GSC ведется в рамках того кванта времени, который выделяется виртуальной машине CP. GSC распределяет обслуживание по принципу абсолютных приоритетов, число градаций приоритета – 256 (приоритет 255 – наивысший). Задача, выбранная на выполнение, вытесняется только тогда, когда она переходит в состояние ожидания или в состояние готовности приходит задача с более высоким приоритетом. Если, однако, имеется несколько задач с одинаковым наивысшим приоритетом, они получают обслуживание в режиме квантования времени. Приоритет задачи/подзадачи устанавливается при ее порождении и может быть изменен только явным образом – системным вызовом CNAR. Механизмы управления задачами в GCS – те же, что и в z/OS:

- системные вызовы ATTACH и DETACH – для порождения и уничтожения задач;
- системные вызовы WAIT и POST и блоки ECB – для синхронизации выполнения;
- системные вызовы ENQ и DEQ – для взаимного исключения доступа к ресурсам.

Linux в z/VM

В разделе, посвященном z/VM будет уместно упомянуть и Linux for 390, и Linux for zSeries. ОС Linux была портирована на мейнфреймы в рамках Advanced Technology Project, и этот проект активно поддерживается IBM. Linux не является чисто гостевой ОС для z/VM, эта ОС может работать и непосредственно на вычислительном комплексе,

однако мощности ОС Linux, разумеется, недостаточно для управления ресурсами полноценного мейнфрейма. Поэтому Linux применяется как самостоятельная ОС в небольшом логическом разделе мейнфрейма или (чаще всего) как гостевая ОС в z/VM. Использование Linux в таком качестве позволяет обеспечить конечных пользователей мейнфрейма рабочими станциями, обладающими гибкостью, надежностью и способностью работать в тесном взаимодействии с другими системами. Немаловажным является то обстоятельство, что виртуальные рабочие станции Linux делают мейнфреймы доступными для огромного числа пользователей, которые не знакомы со спецификой работы в их ОС. Поскольку ОС мейнфреймов поддерживают стандарты работы в Открытой распределенной среде, многие мощные сервисы, обеспечиваемые другими ОС мейнфреймов, являются доступными и для виртуальных рабочих станций Linux.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем состоят достоинства эволюционного подхода IBM к развитию аппаратного и программного обеспечения мейнфреймов?
2. Почему определение "большая ЭВМ универсального назначения" уже не подходит для мейнфрейма?
3. Опишите базовую модель памяти, которая была реализована в ранних версиях VSE. Какой из моделей, описанных нами в главе 3 части I, она соответствует? Как она изменялась в следующих версиях VSE?
4. Сопоставьте способы формирования виртуального адресного пространства в современных версиях VSE и в OS/390. Что между ними общего, в чем различия?
5. Какие дисциплины планирования процессов применяются в VSE?

6. Охарактеризуйте методы доступа BAM и VSAM. Каким образом они совмещаются на одном томе?
7. Какими средствами VSE превращается в интерактивную систему?
8. Опишите модель памяти, которая реализована в MVS – OS/390. Как она изменяется в z/OS?
9. Какие средства взаимодействия процессов имеются в z/OS? Сопоставьте их со средствами, описанными в главе 9 части I.
10. Какие расширения виртуальной памяти сверх первичного адресного пространства может получить приложение в z/OS?
11. Назовите и охарактеризуйте основные подсистемы z/OS, обеспечивающие среды выполнения приложений.
12. Почему автоматизация управления производительностью в z/OS является не роскошью, а необходимостью?
13. Опишите уровни абстракций планирования ресурсов в z/OS. Какими компонентами поддерживается каждый из этих уровней?
14. Назовите основные задачи проекта eLiza. В чем, по-вашему, состоят достоинства проекта?
15. Какие ресурсы предоставляются виртуальной машине в z/VM?
16. Какими методами CP z/VM обеспечивает для виртуальной машины виртуальные устройства?
17. Что такое привилегированные виртуальные машины? Для чего они могут применяться? Какие существуют ограничения на их применение?
18. Сопоставьте алгоритм диспетчеризации z/VM с алгоритмом диспетчеризации VM/370, описанным в разделе 2.3 части I. Какие изменения претерпел этот алгоритм?
19. Опишите политику страничного обмена в z/VM. Каким дисциплинам, описанным в главе 3 части I, она соответствует?

20. Какие гостевые ОС могут работать в среде z/VM? Как их можно классифицировать?
21. Как "распределяются роли" между гостевыми ОС CMS и GCS?
22. Почему гостевой ОС CMS не требуется обеспечивать многозадачность?
23. В чем преимущества файловой системы SFS по сравнению с файловой системой минидисков?
24. ОС Linux для мейнфреймов может работать на реальной ЭВМ, но применяется почти исключительно как гостевая ОС в среде z/VM – почему?

Глава 13. Платформа Java как операционная среда

13.1. Основные свойства платформы Java

Принято считать, что технология Java зародилась в 1980 г. Она была создана группой разработчиков фирмы Sun Microsystems, инициаторами этого проекта являлись Патрик Нотон и Джеймс Гослинг. Первоначально этот проект (тогда он назывался Oak) предназначался для управления включением в сеть бытовых устройств со встроенными вычислительными возможностями. В 1995 году проект получил свое нынешнее название и был переориентирован на программирование в Internet. В дальнейшем возможности и функции языка и платформы Java существенно расширились. На сегодняшний день можно назвать четыре типа программ, создаваемых в рамках технологии Java:

- приложения – программы в обычном смысле, выполняемые, однако, в среде платформы Java;

- апплеты – программы, выполняемые в среде Web-браузера, поддерживающего платформу Java (Sun HotJava, Netscape Communicator, Microsoft Internet Explorer) такие программы могут передаваться по Internet и выполняться на компьютере клиента;
- сервлеты – Java-программы на, которые работают на Web-серверах Java или серверах приложений Java и могут доставлять Web-службы непосредственно в браузер или действовать как промежуточное ПО, которое связывает браузер с серверными службами;
- программы (пока для них нет общего названия), выполняющиеся в средах продуктов промежуточного программного обеспечения, например, программы для сервера приложений Lotus Domino, хранимые процедуры для СУБД IBM DB2 и Oracle и т.п.

Технология Java состоит из двух основных компонентов:

- языка программирования Java [19];
- платформы Java [25].

Язык программирования Java является универсальным объектно-ориентированным языком программирования, синтаксис которого очень похож на синтаксис C++. Отличия Java от C++ состоят в том, что, во-первых, Java гораздо более последовательно воплощает парадигму объектно-ориентированного программирования, во-вторых, в Java отсутствуют некоторые свойства C++, делающие последний трудным для понимания и легким для ошибок (например, арифметика указателей), в-третьих, в Java введены некоторые дополнительные свойства, расширяющие его функциональность (например, нити и синхронизация). Сам по себе язык Java был бы не столь интересен (во всяком случае, для нас), если бы не платформа Java. Платформа Java или среда выполнения Java (JRE – java runtime environment) – это набор программных средств,

обеспечивающих выполнение Java-программы на любой аппаратной платформе и в среде любой ОС. В JRE входит виртуальная машина Java и набор стандартных библиотек Java. Девиз технологии Java – "написано однажды – работает везде". Sun Microsystems декларирует большой набор достоинств языка и платформы Java, но, безусловно, ключевым достоинством Java является переносимость.

Переносимость в Java достигается за счет того, что Java-программа компилируется не непосредственно в команды какой-либо конкретной ЭВМ, а в, так называемый байт-код Java – команды некоторой абстрактной машины, называемой виртуальной машиной Java (Java VM), как показано на рисунке 13.1. Конечным результатом (исполняемым модулем) является файл класса – программа в байт-коде Java. На целевой платформе (на той машине, на которой программа выполняется) должна быть запущена программная Java VM, которая эмулирует ЭВМ, способную выполнять команды байт-кода Java. Сама Java VM – платформенно-зависимая, то есть предназначена для выполнения на конкретной платформе и в конкретной операционной системе. Java VM читает команды байт-кода Java и моделирует их выполнение на той аппаратной платформе и в той операционной среде, в которой она работает. При этом она использует библиотеки Java, также платформенно-зависимые. Стержнем технологии являются спецификации байт-кода Java, файла класса и Java VM. Компиляторы Java могут быть созданы (и создаются) разными разработчиками, но все генерируемые ими исполняемые модули должны соответствовать спецификациям байт-кода Java. Более того, существуют и компиляторы других языков программирования, которые генерируют байт-код Java. Также различными разработчиками могут разрабатываться (и разрабатываются) и Java VM, но все Java VM должны выполнять стандартный байт-код Java.

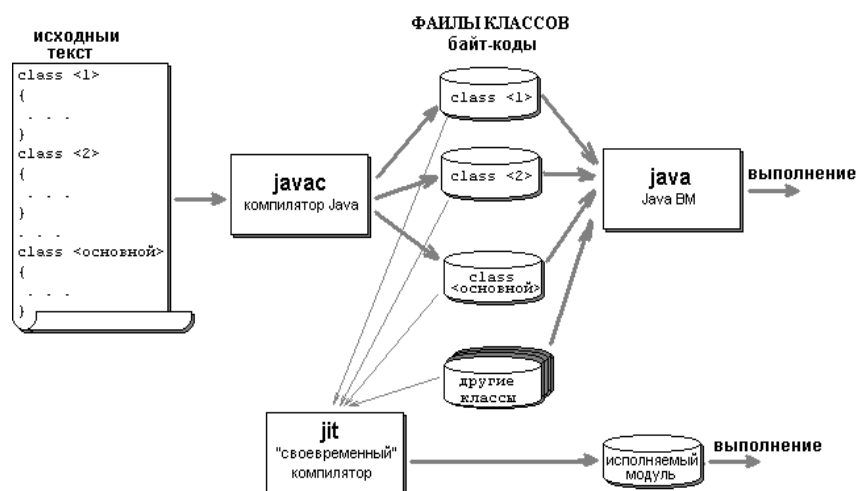


Рисунок 13.1 Выполнение приложения в платформе Java

Итак, Java-программа выполняется в режиме интерпретации. Хотя фирма Sun Microsystems декларирует эффективность в числе основных свойств Java-программ, в отношении быстродействия это утверждение, мягко говоря, сомнительно. Интерпретируемая программа в принципе не может выполняться так же быстро, как программа в целевых кодах. Эффективность работы Java-программ зависит от эффективности работы Java VM, и Java VM разных производителей существенно различаются по этому показателю (лидером является фирма IBM). В составе средств разработки Java имеются также "своевременные" (just-in-time) компиляторы (JIT), которые транслируют байт-код Java в коды целевой платформы, результатом чего является исполняемый модуль в формате целевой платформы и системы. Такой модуль выполняется без участия Java VM, и его выполнение происходит эффективнее, чем выполнение интерпретируемого байт-кода, но это уже выходит за пределы платформы Java.

Таким образом, независимость Java-программ от конкретной аппаратной платформы и ОС достигается за счет того, что Java-платформа является дополнительной "прослойкой" между приложением и ОС и

вместо специфических системных вызовов API конкретной ОС приложение использует API JRE или базовые конструкции языка

Ниже мы рассматриваем некоторые особенности виртуального "процессора" Java VM, как той платформы, на которой выполняются Java-программы.

13.2. Виртуальная машина Java

Типы данных, с которыми работает Java VM, подразделяются на примитивные и ссылочные. Большинство примитивных типов данных Java VM являются также примитивными типами в языке Java. К ним относятся:

- `byte` – 1-байтное целое со знаком;
- `short` – 2-байтное целое со знаком;
- `int` – 4-байтное целое со знаком;
- `long` – 8-байтное целое со знаком;
- `float` – 4-байтное число с плавающей точкой;
- `double` – 8-байтное число с плавающей точкой;
- `char` – 2-байтный символ Unicode.

В отличие от других языков программирования, размеры типов в языке Java и в Java VM являются постоянными, не зависящими от платформы.

Java VM не оперирует типом `boolean`, являющимся примитивным типом языка Java. Для выражений языка, оперирующих этим типом, компилятор Java генерирует коды, оперирующие типом `int`.

Примитивный тип `returnAddress` в Java VM не имеет соответствия в языке Java. Тип `returnAddress` представляет собой указатель на команду байт-кода Java и используется в качестве операнда команд передачи управления.

Ссылочные типы в Java VM и в языке Java являются ссылками (указателями) на объекты – экземпляры классов, массивы и интерфейсы (экземпляры классов, реализующих интерфейсы). Спецификации Java VM не определяют внутренней структуры объектов, в большинстве современных Java VM ссылка на объект является указателем на дескриптор объекта, в котором, в свою очередь содержатся два указателя:

- на объект типа `Class`, представляющий информацию типа, в том числе методы и статические данные класса;
- на память, выделенную для локальных данных объекта в куче.

Все указатели, с которыми работает Java VM, являются указателями в плоском 32-разрядном адресном пространстве, хотя в реализациях Java VM для 64-разрядных платформ могут использоваться и 64-разрядные указатели.

Основные области памяти, с которыми работает Java VM, показаны на рисунке 13.2.

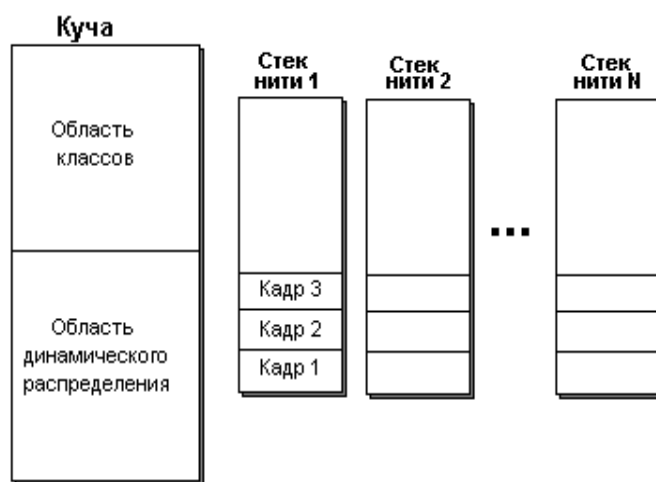


Рисунок 13.2 Основные области памяти Java VM

Область памяти, называемая кучей, разделяется на две части: область классов и область динамически распределяемой памяти (иногда кучей называют только эту часть памяти). Куча создается при запуске Java VM.

Конкретные реализации Java VM могут обеспечивать управление начальным размером кучи и расширение кучи при необходимости.

Класс является основной программной единицей платформы Java, объединяющей в себе данные и методы их обработки. При загрузке класса для него выделяется память в области классов. Каждый класс представляется двумя структурами памяти: областью методов и пулом констант. Область методов содержит исполняемую часть класса – байт-коды методов класса, а также таблицу символических ссылок на внешние методы и переменные. Пул констант содержит литералы класса.

В области динамического распределения выделяется память для размещения объектов. Управление этой областью памяти мы рассматриваем в отдельном разделе.

Java VM поддерживает параллельное выполнение нескольких нитей. Для каждой нити при ее создании Java VM создает набор регистров и стек нити.

Набор регистров включает в себя четыре 32-разрядных регистра:

- `pc` – регистр-указатель на команду;
- `optop` – регистр-указатель на вершину стека операндов текущего кадра;
- `var` – регистр-указатель на массив локальных переменных текущего кадра;
- `frame` – регистр-указатель на среду выполнения текущего метода.

Стек нити представляет собой стек в традиционном понимании, то есть списковую структуру данных, обслуживаемую по дисциплине "последним пришел – первым ушел". Элементами стека являются кадры (frame) методов. В традиционных блочных языках программирования при помощи стека обеспечиваются вложенные вызовы процедур. Аналогичным

образом Java VM через стек нити обеспечивает вложенные вызовы методов, представляя каждый метод кадром в стеке. Новый кадр создается и помещается в вершину стека при вызове метода. Кадр, расположенный в вершине стека является текущим, он соответствует методу, выполняемому в нити в текущий момент. При возврате из метода его кадр удаляется из стека. Управление начальным размером стека нити и возможность его динамического расширения зависит от реализации Java VM. Спецификации Java VM не требуют размещения стека нити в непрерывной области памяти.

Кадр, как было сказано, создается динамически и содержит три основных области.

- набор локальных переменных экземпляра класса, на который ссылается регистр `var`;
- стек операндов, на который ссылается регистр `optr`;
- структуры среды выполнения, на которую ссылается регистр `frame`.

Эти области показаны на рисунке 13.3.

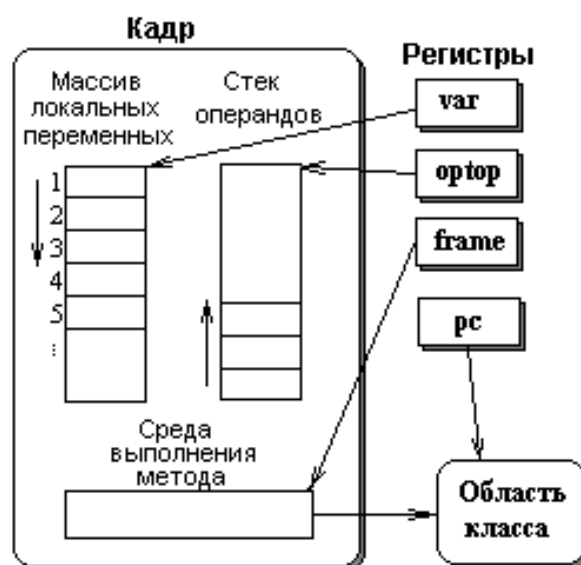


Рисунок 13.3 Структура и связи кадра

Набор локальных переменных представляет собой массив 32-разрядных слов. Данные двойной точности (типы `long` и `double`) занимают по два смежных слова в этом массиве. Размер этого массива – фиксирован для метода, так как число локальных переменных метода становится известным уже на этапе компиляции. Операнды команд байт-кода, которые оперируют локальными переменными, представляются индексами в этом массиве.

Java VM является стековой машиной. Это означает, что в ней нет регистров общего назначения, и операции производятся над данными, находящимися в стеке. Этой цели служит стек операндов, выделяемый в составе каждого кадра. При выполнении команд байт-кода Java, изменяющих данные, операнды таких команд выбираются из стека операндов, в тот же стек помещаются и результаты выполнения команд.

Среда выполнения метода содержит информацию, необходимую для динамического связывания, возврата из метода и обработки исключений. Код класса (размещенный в области класса) обращается к внешним методам и переменным, используя символические ссылки. Динамическая компоновка переводит символические ссылки в фактические. Среда выполнения содержит ссылки на таблицу символов метода, через которую производятся обращения к внешним методам и переменным.

В среде выполнения содержится также информация, необходимая для возврата из метода: указатель на фрейм вызывающего метода, значение регистра `pc` для возврата, содержимое регистров вызывающего метода и указатель на область для записи возвращаемого значения.

Информация обработки исключений содержит ссылки на секции обработки исключений в методе класса.

Через среду выполнения также происходит обращения к данным, содержащимся в области класса, в том числе к константам и к переменным класса.

Команды Java VM состоят из однобитного кода операции, а также могут содержать операнды. Число и размер операндов определяются кодом операции, некоторые команды не имеют операндов. Основной алгоритм работы Java VM сводится к простейшему циклу, приведенному на рисунке 13.4.



Рисунок 13.4 Основной цикл работы Java VM

Каждый из типов данных Java VM обрабатывается своими командами. Основные типы команд Java VM:

- Команды загрузки и сохранения, в том числе:
 - загрузка в стек локальной переменной;
 - сохранение значения из стека в локальной переменной;
 - загрузка в стек константы (из пула констант).
- Команды манипулирования значениями (большинство этих операций работают с операндами из стека и помещают результат в стек), в том числе:

- арифметические операции;
- побитовые логические операции;
- сдвиг;
- инкремент (операция работает с операндом – локальной переменной).
- Команды преобразования типов.
- Команды создания ссылочных данных и доступа к ним, в том числе:
 - создания экземпляров класса;
 - доступа к полям класса;
 - создания массивов;
 - чтения в стек и сохранения элементов массивов;
 - получения свойств массивов и объектов.
- Команды прямого манипулирования со стеком.
- Команды передачи управления, в том числе:
 - безусловный переход;
 - условный переход;
 - переход по множественному выбору.
- Команды вызова методов и возврата (включая специальные команды вызова синхронизированных методов).
- Команды генерации и обработки исключений.

Принятые в спецификациях Java ВМ структуры данных и алгоритмы таковы, что позволяют реализовать виртуальную машину с минимальными затратами памяти и сделать ее работу максимально эффективной.

Другим ключевым элементом спецификаций Java является файл класса. Каждый файл класса описывает один класс или интерфейс. Файл класса содержит поток байт, структурированный определенным образом. Все реализации компилятора Java должны генерировать файлы классов, структура которых соответствует определенной в спецификациях. Все

реализации Java VM должны "понимать" структуру файлы класса, соответствующую определенной в спецификациях.

Основные компоненты файла класса следующие:

- Некоторая верификационная информация: "магическое число" – сигнатура файла класса, номер версии.
- Флаг доступа, отображающий модификаторы, заданные в определении класса (`public`, `final`, `abstract` и т.д.), а также признак класса или интерфейса.
- Пул констант – таблица структур, представляющие различные строковые константы – имена классов и интерфейсов, полей, методов и другие константы, на которые есть ссылки в файле класса.
- Ссылки на имена `this`-класса и суперкласса в пуле констант.
- Перечень интерфейсов, реализуемых классом (в виде ссылок в пул констант).
- Описание полей класса с указанием их имен, типов, модификаторов и т.д.
- Методы класса – каждый метод представляется в виде определенной структуры, в которой содержится описание метода (имя, модификаторы, и т.д.), одним из атрибутов этой структуры является массив байт-кодов метода.

Многие компоненты файла класса (пул констант, перечень интерфейсов и др.) имеют нефиксированную длину, такие компоненты предваряются 2-байтным полем, содержащим их длину.

13.3. Многопоточность и синхронизация

Java, по-видимому, является единственным универсальным языком программирования, в котором механизмы создания нитей поддерживаются встроенными средствами языка. В традиционных языках программирования (например, C) создание нитей обеспечивается системно-зависимыми библиотеками, обеспечивающими API ОС. В Java средства создания нитей системно-независимые.

В Java-программе нить представляет отдельный класс, который может быть создан

- либо как подкласс (наследник) суперкласса `Tread`;
- либо как класс, реализующий интерфейс `Runnable`, внутри этого класса должна быть переменная экземпляра класса – ссылка на объект класса `Tread`.

Суперкласс `Tread` и интерфейс `Runnable` определены в базовой библиотеке языка Java – пакете `java.lang`. При любом варианте создания в классе-нити должен быть реализован метод `run()`. Выполнение метода `start()` для экземпляра такого класса вызывает выполнения метода `run()` в отдельном потоке вычисления.

Как мы увидели в предыдущем разделе, Java VM обеспечивает для каждой нити собственную среду вычисления – собственный набор регистров и стек (в некоторых реализациях Java VM обеспечивает для нити также и собственную кучу).

Но Java VM не выполняет действий по планированию нитей на выполнение. Для этого библиотечные методы Java обращаются к ОС, используя API той ОС, в среде которой работает Java VM.

Для нитей в Java предусмотрено управление приоритетами (методы `getPriority()`, `setPriority()`), однако, и здесь Java использует механизмы управления приоритетами ОС. Так, уже классическим для учебников по Java является пример апплета, в котором по экрану

"наперегонки" движутся несколько объектов, движение каждого осуществляется в отдельной нити и с собственным приоритетом. Этот пример весьма наглядно демонстрируется в средах, например, OS/2 и Linux, но выглядит не очень убедительным в среде Windows 95/98, так как приоритет нити не слишком влияет на скорость движения объекта – примерно так, как показано на рисунке 13.5.

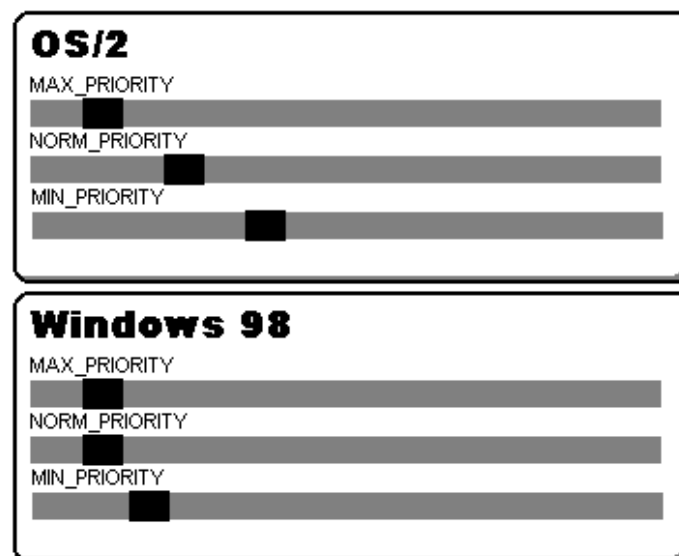


Рисунок 13.5 "Гонки" в разных операционных средах (движение справа налево).

Любая нить может быть сделана нитью-"демоном". Нить-"демон" продолжает выполняться даже после окончания той нити, в которой она была создана. Нить становится "демоном" при ее создании только в том случае, если она создается из нити-"демона". Программа может изменить состояния нити при помощи метода `setDaemon()` класса `Thread`. Выполнение любой программы Java VM начинается с единственной нити (в которой вызывается метод `main()`), и эта нить запускается не как "демон". Java VM продолжает существовать, пока не завершатся все нити не-"демоны".

В языке Java имеется также класс `ThreadGroup` – группа нитей, которая может управляться совместно.

Если в Java предусмотрены нити, то, естественно, должны быть предусмотрены и средства синхронизации и взаимного исключения при параллельной работе нитей. Основным средством синхронизации и взаимного исключения в Java является ключевое слово `synchronized`, которое может употребляться перед каким-либо программным блоком. Ключевое слово `synchronized` определяет невозможность использования программного блока двумя или более нитей одновременно. В Java `synchronized`-блоки называются мониторами и их фактическая тождественность мониторам Хоара, описанным в разделе 8.8 части I, очевидна. В зависимости от деталей способа употребления `synchronized` может работать как:

- защищенная (`guard` – см. раздел 8.8 части I) процедура – в том случае, если `synchronized`-блок представляет собой целый метод, однако, в отличие от описанных нами `guard`-процедур одновременное вхождение в разные `synchronized`-методы возможно;
- анонимные скобки критической секции – в том случае, если `synchronized`-блок является просто программным блоком;
- скобки критической секции с защитой выбранного ресурса – в том случае, если после ключевого слова `synchronized` указывается в скобках ссылка на объект.

В первоначальной версии языка Java для класса `Thread` предусмотрены методы:

- `resume()` – приостановить выполнение нити;
- `suspend()` – возобновить выполнение нити;
- `yield()` – сделать паузу в выполнении нити, чтобы дать возможность выполниться другой нити;
- `join()` – ожидать завершения нити.

Эти средства позволяют синхронизировать работу нитей, но в следующих версиях был (наряду со старыми средствами) введен новый, более стройный аппарат синхронизации и взаимного исключения.

Класс `Object` имеет три метода:

- `wait()` – ожидать уведомления об этом объекте;
- `notify()` – послать уведомление одной из нитей, ждущих уведомления об этом объекте;
- `notifyAll()` – послать уведомление всем из нитям, ждущим уведомления об этом объекте.

Поскольку класс `Object` является корнем иерархии классов, объекты всех – стандартных и пользовательских – классов являются его подклассами и наследуют эти методы. Эти методы аналогичны операциям `wait` и `signal`, описанным нами в разделе 8.7 части I. Реализация, например, общего (с возможным значением, большим 1) семафора с использованием этих средств будет выглядеть следующим образом:

```
/***/ семафор реализуется
//      в виде класса Semaphore
public class Semaphore
{
    // значение семафора
    private int Semaphore_value;
    /**/ пустой конструктор семафора, по умол-
    // чанию начальное значение семафора – 0
    public Semaphore()
    {
        this(0);
    }
}
```

```

/** конструктор с параметром -
// начальным значением семафора,
// если задано отрицательное значение,
// устанавливается начальное значение 0
public Semaphore(int val)
{
    if (val < 0) Semaphore_value = 0
    else Semaphore_value = val;
}

/** V-операция
// V- и P-операции объявлены synchronized,
// чтобы исключить одновременное выполне-
// ние их двумя или более нитями
public synchronized void V()
{
    // возможно пробуждает нить,
    // ожидающую у семафора
    if (Semaphore_value == 0) this.notify();
    // увеличивает значение семафора
    Semaphore_value ++;
}

/** P-операция
public synchronized void P()
    throws InterruptedException
    // исключение может выбрасываться в wait
{
    // если значение семафора равно 0,
    // блокирует нить
    while (counter == 0) this.wait();
}

```

```
// уменьшает значение семафора
Semaphore_value --;
}
}
```

В Java VM с каждым объектом связывается замок (lock) и список ожидания (wait set).

В спецификациях байт-кода Java имеются специальные команды `monitorenter` и `monitorexit`, устанавливающие и снимающие замок. Java VM, входя в `synchronized`-блок, пытается выполнить операцию установки замка и не продолжает выполнения нити, пока операция не будет выполнена. При выходе из `synchronized`-блока выполняется операция снятия замка.

Список ожидания используется методами `wait()`, `notify()`, `notifyAll()`. Он представляет собой список нитей, ожидающих уведомления о данном объекте. Названные операции работают с этим списком очевидным образом.

Следует отметить, что многопоточность, заложенная в языке Java, имеет большие перспективы. Изначально сама идея нитей (а ее первая коммерческая реализация была сделана именно фирмой Sun Microsystems) возникла как решение, призванное обеспечить эффективную загрузку оборудования в многопроцессорных системах, но теперь свойства многопоточности, закладываемые в программное обеспечение, оказывают (наряду с другими факторами) обратное влияние на процессорные архитектуры, стимулируя развитие в них средств распараллеливания вычислительного процесса.

Так, новый проект компьютерной архитектуры фирмы Sun Microsystems носит название MAJC (Microprocessor Architecture for Java Computing – архитектура микропроцессора для Java-вычислений), которое

говорит само за себя. В концепцию этой архитектуры (как, впрочем, и ряда других новых архитектур) входит многопроцессорная обработка с несколькими "потокowymi устройствами" в каждом процессоре. Такая архитектура призвана обеспечить более эффективную обработку на сетевом сервере современных потоков данных, которые характеризуются возрастанием удельного веса в них мультимедийной информации.

Для получения лучшей производительности, кроме многопоточных микропроцессоров, разработчики MAJS применяют технологию, называемую "пространственно-временными вычислениями" или "предположительной многопоточности". В этой технологии прикладной программист вообще не будет беспокоиться о распараллеливании своих программ, потому что эта работа будет сделана за него Java VM.

Смысл пространственно-временных вычислений сводится к следующему. Java VM проверяет программу и предполагает, что два метода могут выполняться одновременно на двух процессорах. Она посылает метод А на первый процессор, а метод В – на второй для предположительного вычисления. Поскольку В – предположительный метод, он выполняется в отдельном адресном пространстве, называемом предположительной памятью. Если все идет хорошо, и никакие зависимости в данных не нарушены, то предположительная память сливается с основной памятью и программа обрабатывает следующую пару методов. Если произошло нарушение, то второй метод отменяется и предположительная память "выбрасывается".

Идея пространственно-временных вычислений не является кардинально новой, но она не применялась в обычных многопроцессорных системах без многопоточности, так как позволяла увеличить эффективность выполнения программ в 2-процессорной конфигурации не более, чем на 5%. Разработчики MAJS рассчитывают, что в их архитектуре

производительность последовательных приложений на двух процессорах должна возрасти в 1.6 раза.

13.4. Управление памятью в куче

Управление памятью относится к числу тех свойств языка Java, которые заложены в само его ядро и непосредственно обеспечиваются Java VM. Особенностью управления памятью в Java является то, что с точки зрения прикладного программиста его практически нет. Память для данных примитивных типов выделяется в области локальных переменных кадра. Кадр для метода выделяется только на время выполнения метода, при завершении выполнения память кадра, а, следовательно, и все локальные переменные освобождается. Этот механизм подобен размещению локальных переменных в стеке в традиционных блочных языках программирования (C/C++, PL/1 и т.д.). Ссылочные типы состоят из двух частей: ссылки на объект и собственно тела объекта. Массивы в Java также являются ссылочным типом, и все, что далее говорится про объекты, справедливо и для массивов. Ссылка представляет собой адрес памяти, указатель на объект в терминах языка C/C++, но в отличие от C/C++ адресная арифметика в Java не разрешена. Объект в программе доступен только через переменную, являющуюся ссылкой на него.

Итак, память, выделяемая для ссылок, управляется автоматически, как и память для примитивных типов. Иначе обстоит дело с памятью, выделяемой для тела объекта. В языке Java имеется операция `new`, которая явным образом выделяет память для тела объекта и возвращает ссылку на созданный объект. Память для каждого объекта выделяется явным образом, при помощи этой операции. Создание новых объектов возможно

также неявным образом – некоторые библиотечные методы создают (при помощи той же операции `new`) новый объект и возвращают ссылку на созданный объект. На этом "заботы" прикладной Java-программы об управлении памятью заканчиваются. Программа не освобождает выделенную память, это делает за нее Java VM. Автоматическое освобождение памяти, занимаемой уже ненужными (неиспользуемыми) объектами, – одна из наиболее интересных особенностей платформы Java. Это освобождение выполняется в Java VM программным механизмом, который называется сборщиком мусора (`garbage collector`). Но что такое неиспользуемый объект? Программа может "оставить объект в покое" на долгое время, а потом вдруг вновь вернуться к нему. Время обращения к объекту (как это делается в дисциплине управления памятью LRU) не может служить показателем ненужности объекта. Сборщик мусора считает неиспользуемыми те объекты, на которые нет ссылок. Если в программе нет ссылки на объект, то программа принципиально не может обратиться к объекту, следовательно, объект представляет собой мусор. Обратите внимание на то обстоятельство, что при выходе из блока, в котором был создан объект, освобождается память, занимаемая ссылкой на объект, но это еще не значит, что объект сразу же становится мусором. Ссылка на созданный объект может быть присвоена внешней по отношению к данному блоку переменной или быть возвращаемым значением метода. Если же этого не происходит, то объект действительно становится мусором. При выполнении Java-программы такой мусор в памяти накапливается. Многие методы библиотечных классов Java (например, класса `String`) построены таким образом, что их использование способствует интенсивному накоплению мусора в памяти.

Когда накопление мусора приводит к нехватке памяти, вступает в действие сборщик мусора. Для обеспечения работы сборщика мусора в дескрипторе каждого объекта имеется "признак мусора". При создании

объекта "признак мусора" устанавливается во взведенное состояние. Алгоритм работы сборщика мусора (один из его вариантов) состоит из двух фаз:

Фаза маркировки. Сборщик мусора просматривает области локальных переменных всех активных в настоящий момент методов, а также поля всех доступных объектов. В дескрипторах тех объектов, на которые есть ссылки в просмотренных областях "признак мусора" сбрасывается

Фаза очистки. Просматривается область кучи, дескрипторы всех объектов. Те объекты, "признак мусора" которых оказывается взведенным (не был сброшен в фазе маркировки), являются мусором, занимаемая ими память освобождается. У тех же объектов, "признак мусора" которых сброшен, этот признак взводится – для подготовки к следующей сборке мусора.

Затраты на выполнение сборки мусора практически не зависят от количества мусора – в любом случае требуется полный просмотр и областей локальных переменных, и кучи. Следовательно, сборку мусора выгоднее производить только в те моменты, когда мусора накопится много: в этом случае при тех же затратах будет получен больший результат. Поэтому операция сборки мусора может создавать некоторую проблему при выполнении Java-программ. Проблема состоит в том, что момент активизации сборщика мусора непредсказуем, а когда такая активизация произойдет, она вызовет задержку в вычислениях. В новых реализациях Java VM эту проблему стараются если не решить кардинально, то несколько сгладить, запуская сборщик мусора в отдельной низкоприоритетной нити.

Хотя область методов тоже формально принадлежит куче, в большинстве современных Java VM сборщик мусора в этой области не работает.

13.5. Защита ресурсов

Поскольку одной из основных сфер применения технологии Java является Internet, вопросы безопасности для этой технологии приобретают особое значение. Безопасность в сетевой среде представляет собой целый комплекс сложных вопросов, рассматриваемых в отдельном курсе. Здесь же мы уделим основное внимание защите локальных ресурсов – анализу возможности Java-программы получить несанкционированный доступ к ресурсам на том компьютере, на котором она выполняется.

Прежде всего, в самих языковых средствах Java отсутствуют некоторые возможности языка C/C++, которые наиболее часто приводят к неправильному использованию ресурсов – случайному или намеренному. Главная черта языка Java в этом отношении – отсутствие указателей. Хотя доступ к объектам в Java осуществляется по ссылкам, и физический смысл ссылки и указателя C/C++ одинаков – адрес памяти, ссылка не есть указатель. Различие состоит в том, что, во-первых, ссылка не может быть преобразована в число или какое-либо иное представление физического адреса, во-вторых, над ссылками недопустимы арифметические операции. Именно адресная арифметика в C/C++ является средством, использование которого может привести к доступу процесса за пределы той области памяти, к которой он имеет право обращаться.

Другой "лазейкой" для выполнения несанкционированных действий в языке C/C++ является слабая защита типов. C/C++ позволяют использовать типы данных в операциях, этому типу не свойственных – путем неявного преобразования типов или путем приравнивания разнотипных указателей (в том числе, и для интегрированных типов). В Java осуществляется строгий контроль типов и в большинстве случаев требуется явное преобразование типов.

Автоматическое освобождение памяти в Java также является свойством, повышающим защищенность. Можно говорить также и о том, что более последовательное воплощение в Java парадигмы объектно-ориентированного программирования также является выигрышным с точки зрения защиты обстоятельством.

Следует оговорить, что указанные различия между языками C/C++ и Java обусловлены прежде всего тем, что языки ориентированы на разные сферы применения. Те "недостатки" языка C/C++, на которые мы указываем, превращаются в уникальные достоинства при применении C/C++ в качестве языка системного программирования, а именно таково первоначальное предназначение этого языка. При разработке же приложений (а Java – язык именно для разработки приложений) эти возможности становятся ненужными и даже опасными.

Однако сами свойства языка Java еще не являются гарантией защищенности. Они обеспечиваются компилятором Java, но не предохраняют от модификации исполняемый модуль. Поскольку спецификации байт-кода Java и файла класса открыты, программы, осуществляющие несанкционированный доступ, могут писаться непосредственно в байт-кодах или на других языках с компиляцией в байт-код Java. Чтобы перекрыть этот канал несанкционированного доступа, в платформе Java выполняется верификация байт-кода.

Процесс верификации состоит из четырех шагов.

Шаг 1 выполняется при загрузке класса. При этом Java VM проверяет базовый формат файла класса – "магическое число" и номер версии, соответствие размера файла суммарному размеру его составляющих, формальное соответствие отдельных структур спецификациям.

Шаг2 выполняется при связывании, он включает в себя верификацию без анализа байт-кодов. На этом шаге проверяется:

- отсутствие нарушений в использовании классов и методов, объявленных с модификатором `final`;
- наличие у каждого класса (кроме класса `Object`) суперкласса;
- соответствие спецификациям содержимого пула констант;
- правильность имен классов и интерфейсов и дескрипторов всех полей и методов, ссылающихся на пул констант.

Проверки правильности элементов файла класса, выполняемые на этом шаге, – только формальные, не семантические. Более подробные проверки выполняются на следующих шагах.

Шаг 3 также выполняется на этапе связывания. На этом шаге верификатор проверяет массив байт-кодов каждого метода. При этом анализируется поток данных, обрабатываемый при выполнении метода. Верификатор исходит из того, что в любой точке программы, независимо от того, каким образом управление попало на эту точку должны соблюдаться определенные ограничения целостности данных, которые сводятся в основном к следующим;

- размер стека операндов неизменен и стек содержит операнды одного типа;
- не выполняется доступ к локальным переменным неизвестного типа;
- доступ к локальным переменным осуществляется только в пределах массива локальных переменных;
- все обращения к пулу констант производятся к элементам соответствующего типа;
- полям класса назначаются значения соответствующего типа;
- все команды байт-кода используются с операндами (в стеке или в массиве локальных переменных) типа, соответствующими типу команды;

- методы вызываются с правильными аргументами;
- команды перехода передают управление только внутри байт-кода метода и передача управления всегда происходит только на первый байт команды байт-кода.

Шаг 4 выполняется при первом вызове кода любого метода. Это "виртуальный шаг", он выполняется не в виде отдельного шага проверки всего байт-кода, а при выполнении каждой отдельной команды.

Для команды, которая ссылается на тип, при этом:

- загружается определение типа (если оно еще не загружено);
- проверяется, может ли текущий выполняемый метод ссылаться на этот тип;
- выполняется инициализация класса (если он еще не инициализирован).

Для команды, которая вызывает метод или осуществляет доступ к полю класса, при этом:

- существует ли поле или метод в данном классе;
- проверяется правильность дескриптора вызванного метода или поля;
- проверяется, имеет ли текущий выполняемый метод права доступа к этому методу или полю.

В конкретных реализациях Java VM допускается после выполнения шага 4 заменять проверенную команду байт-кода альтернативной "быстрой" формой. Например, в Sun Java VM команда байт-кода `new` может быть заменена командой `new_quick`. "Быстрая" команда выполняется так же, как и исходная, но при ее выполнении исключается повторная верификация команды. В файле класса "быстрые" команды не допускаются, они выявляются на предыдущих шагах верификации и

вызывают отказ. "Быстрые" формы не являются спецификациями Java, они реализуются в конкретной Java VM.

Апплеты являются наиболее критическими с точки зрения безопасности Java-программами, поскольку апплет загружается из Internet, возможно, из непроверенного источника. Естественно, недопустимым является предоставление программе, пришедшей "неизвестно откуда" доступа к ресурсам локального компьютера. Поэтому для апплетов введены весьма жесткие ограничения на выполнение. Апплету запрещается:

- получать сведения о пользователе или его домашней директории;
- определять свои системные переменные;
- работать с файлами и директориями на локальном компьютере (читать, изменять, создавать и т.д. и даже проверять существование и параметры файла);
- осуществлять доступ по сети к удаленному компьютеру, получать список сетевых сеансов связи, которые устанавливает локальный компьютер с другими компьютерами;
- открывать без уведомления новые окна, запускать локальные программы и загружать локальные библиотеки, создавать новые нити, получать доступ к группам нитей другого апплета;
- получать доступ к любому нестандартному пакету, определять классы, входящие в локальный пакет.

Модель безопасности Java еще далека от совершенства, и в ее реализациях иногда обнаруживаются "лазейки" для несанкционированного проникновения. Следует отметить, что в сетевых публикациях довольно часто можно встретить критику безопасности в Java и предупреждение о принципиальной возможности "взлома" защиты Java тем или иным способом. Вместе с тем, сетевые публикации не дают оснований говорить о том, что реальные информационные системы, в которых применяется

технология Java, чаще подвергаются взлому, чем системы, эту технологию не применяющие.

13.6. JavaOS и Java для тонких клиентов

В конце 90-х годов фирма Sun Microsystems предприняла разработку новой ОС, базирующейся на технологии Java – JavaOS. Для доводки этой ОС фирма Sun привлекла фирму IBM, и конечный продукт JavaOS является собственностью обеих этих фирм.

JavaOS является операционной системой для широкого спектра вычислительных средств, включая сетевые и встроенные компьютеры. Целью разработки этой ОС являлось предоставление среды для выполнения Java-приложений без использования базовой универсальной ОС.

JavaOS строится по принципу многослойной архитектуры, показанной на рисунке 13.6, включающей в себя платформенно-зависимую и платформенно-независимую части.

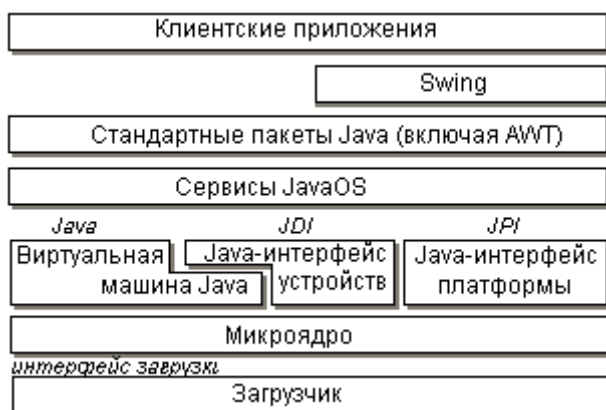


Рисунок 13.6 Архитектура JavaOS

Платформенно-зависимая часть состоит из загрузчика, микроядра, виртуальной машины Java и частично – среды выполнения Java (JavaOS Runtime Environment).

Функции загрузчика вытекают из его названия. JavaOS ориентирована прежде всего на клиент/серверную модель вычислений. Это означает, что необходимое программное обеспечение, постоянно хранящееся на клиентской стороне – минимальное, загрузчик и составляет тот необходимый и достаточный минимум программного обеспечения, который обеспечивает загрузку всего остального программного обеспечения с сервера.

Микроядро JavaOS очень похоже на микроядра ОС, рассмотренных нами выше (QNX, AMX RTOS и др.). Оно выполняет функции:

- обработки прерываний и исключений;
- поддержки множественных нитей;
- поддержки многопроцессорных конфигураций;
- управления реальной памятью;
- управления реальными устройствами и каналом ПДП.

JVM обеспечивает:

- интерпретацию байт-кода;
- управление выполнением;
- управление памятью;
- нити;
- загрузку классов;
- верификацию байт-кода.

Программное обеспечение среды выполнения Java частично создается в кодах Java, частично – в "родных" (native) кодах целевой платформы. В состав среды выполнения входит JVM и ряд системных менеджеров, в том числе:

- Менеджер Конфигурации, представляющий собой первый класс, Java-кода, выполняемый JVM, он обеспечивает запуск компонентов Менеджера Платформы и дополнительных сервисов JavaOS;
- Менеджер Платформы, обеспечивающий запуск и поддержку компонентов, обслуживающих платформенно-зависимые устройства и шину ввода-вывода платформы;
- Менеджер Сервисов, пакет, обеспечивающий поиск и запуск сервисных утилит JavaOS;
- Менеджер Устройств, компонент, обеспечивающий архитектуру Java-интерфейса устройств (JDI);
- Классы Java-интерфейса платформы (JPI), инкапсулирующие драйверы JDI и решение платформенно-зависимых вопросов, включая управление временем, памятью и прерываниями.

Дополнительные (опциональные) компоненты среды выполнения включают в себя компоненты конфигурации (персональной, сетевой, встроенной), наборы драйверов и средства отладки.

Вся среда выполнения (включая JVM) работает как один процесс в виртуальном адресном пространстве. Соответствие виртуального адресного пространства физической памяти обеспечивается микроядром. Также микроядро обеспечивает использование многопроцессорной архитектуры вычислительной системы для функционирования среды выполнения и приложений. Вся специфика управления процессорами и памятью инкапсулирована в JPI.

Большая часть драйверов устройств JavaOS пишется на языке Java. Платформенная независимость драйверов поддерживается компонентом JDI, который состоит из:

- Менеджера Событий, обеспечивающего взаимодействие с устройствами по событийной модели;
- Системной Базы Данных, обеспечивающей хранение и получение конфигурационной информации (относящейся к ОС, устройствам и приложениям) в едином репозитории;
- платформенно-зависимых блоков драйверов;
- Менеджера Шины.

Следующий, полностью платформенно-независимый уровень составляют сервисы JavaOS, такие как: классы, обеспечивающие базовую графику, ввод-вывод и сетевые коммуникации для платформы.

Более высокие уровни составляют стандартные пакеты Java, пакет расширенного графического интерфейса Swing и, наконец, пользовательские приложения.

К сожалению, JavaOS "не успела" на рынок тонких клиентов, к тому моменту, когда эта ОС поступила в продажу, рынок мобильных клиентов, на который она могла претендовать, был уже занят, в основном, Windows CE, также сложились уже и операционные среды для сетевых компьютеров, например, IBM Workspace on Demand для OS/2 и Windows. Поэтому фирмы-производители "законсервировали" проект и его конечный продукт – JavaOS – не представлен на рынке.

Опыт разработки JavaOS фирма Sun Microsystems использовала для создания концепции EmbeddedJava [17]. Технология EmbeddedJava является надстройкой над ОС (любой ОС) тонкого клиента и включает в себя JVM и библиотеку классов Java. Отличие от базовой технологии Java состоит в том, что и JVM, и библиотека классов являются конфигурируемыми, т.е. их объем минимизируется таким образом, чтобы в них включались только те свойства, которые необходимы и достаточны для выполнения Java-приложений конкретного тонкого клиента. Фирма

Sun обеспечивает набор инструментальных средств для создания такой компактной прикладной среды, в состав которых входят:

- **JavaFilter** – инструмент для выявления тех классов, полей и методов, которые необходимы для функционирования приложения
- **JavaCodeCompact** – инструмент для оптимизации кода приложения для экономии RAM- и ROM-памяти;
- **JavaDataCompact** – инструмент для компактного представления внешних по отношению к приложению данных.

После определения необходимых компонент и создания компактных кодов и данных приложение совместно с компонентами среды выполнения компилируется в коды целевой платформы (native-коды), которые могут быть помещены в RAM- или ROM-память "тонкого" устройства. Общий ход процесса разработки приложения EmbeddedJava показан на рисунке 13.7.

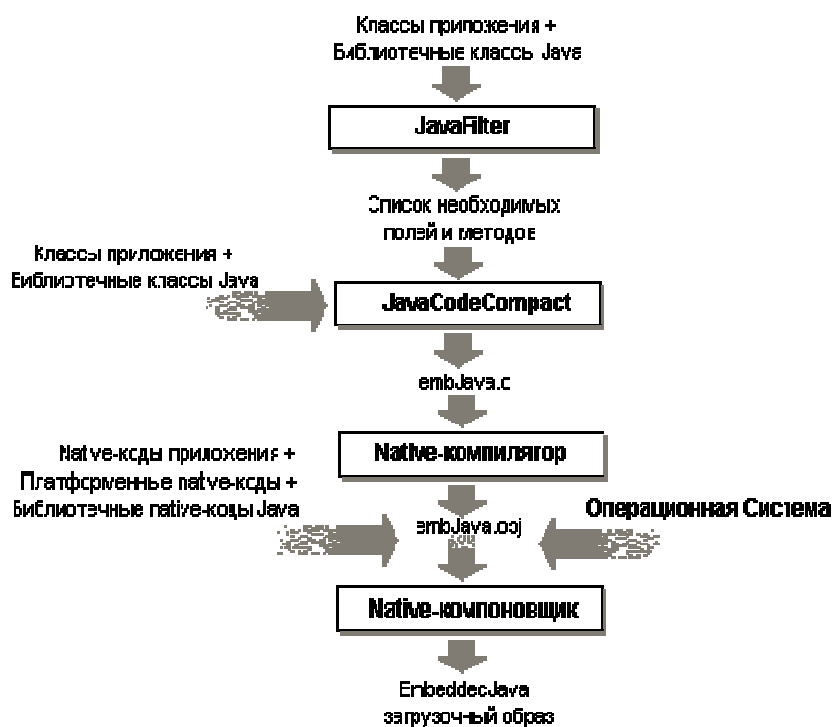


Рисунок 13.7 Процесс разработки приложения EmbeddedJava

13.7. Перспективы технологий Java

Технологии Java были на подъеме несколько последних лет, можно предполагать, что их интенсивное развитие и влияние на другие информационные технологии – явление долговременное. В настоящее время стандарты технологий Java открыты, в их развитии, наряду с Sun Microsystems, активно участвуют ведущие производители рынка информационных технологий (IBM, Hewlett-Packard, Oracle и другие). Фирма Microsoft, хотя и производит конкурирующие технологии, также вынуждена считаться с технологиями и стандартами Java. В условиях распространения среды сетевых вычислений технология Java является одним из главных средств обеспечения совместной работы в глобальном информационном пространстве аппаратных и программных средств от разных производителей. Другим таким средством, обеспечивающим совместимость по данным, является язык XML, также родившийся в Sun Microsystems, с которым Java сейчас интегрируется (стандарт Java Standard Extension for XML).

Первый успех технологий Java был обеспечен прежде всего апплетами, то есть программами, выполняющимися на удаленном клиенте. Нынешнее же развитие и перспективы этих технологий связаны в основном с серверным программным обеспечением. Основные стандарты этого направления: Enterprise JavaBeans (EJB) – компонентная архитектура построения расширяемых, многоуровневых, распределённых приложений для серверов и Java 2 Platform, Enterprise Edition (J2EE), расширение возможностей межплатформенной переносимости EJB. Важной является также достигнутая совместимость Java с реляционными базами данных (стандарты SQLJ и JDBC, развиваемые под эгидой ISO).

Развитие "тонких" клиентов сети заставляет технологии Java вновь обратиться и "истокам" (проект Oak) – построению программного

обеспечения для неполнофункциональных клиентских устройств. Поскольку "тонкие" клиенты отличаются большим разнообразием аппаратных и программных платформ, именно Java может стать той технологией, которая позволит таким клиентам интегрироваться в глобальное информационное пространство. Технология Jini, которая базируется на Java-технологии, позволяет работать с любыми устройствами и отказаться от традиционного использования разнообразных драйверов, громоздкого системного программного обеспечения, привязанного к аппаратным платформам и не позволяющего устройствам взаимодействовать в гетерогенной сети. Jini – это сетевая инфраструктура, набор соглашений специфицирующих методы автоматического взаимодействия и регистрации устройств, подключаемых к сети.

Таким образом, не исключено, что ближайшие годы развития информационных технологий пройдут "под знаменем" технологий Java.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Чем обеспечивается переносимость Java-программ?
2. Из каких ключевых элементов состоит технология Java?
3. Каковы основные "аппаратные" компоненты Виртуальной Машины Java?
4. Какие структуры памяти создает Виртуальная Машина Java?
5. В чем состоит специфика выделения/освобождения памяти в Java? Как она обеспечивается Виртуальной Машиной Java?
6. Какие принципиальные отличия многопоточности языка Java от многопоточности, например, языка C/C++?

7. Какие из средств синхронизации и взаимного исключения, описанных в главе 8 части I, могут быть реализованы в Java при помощи модификатора `synchronized`?
8. Какие методы синхронизации имеются в Java 1.0? какие – в Java 1.1 и далее?
9. Какие средства, обеспечивающие защиту ресурсов, заложены в сам язык Java? Какие средства обеспечиваются Виртуальной Машиной Java?
10. Что представляют собой апплеты Java с точки зрения защиты ресурсов?
11. В чем сходство JavaOS с другими рассмотренными нами ОС на базе микроядра? в чем отличия?
12. Каковы особенности технологий Java фирмы Sun для тонких клиентов?

Заключение

В настоящее время основная борьба на рынке операционных систем разворачивается между различными версиями Windows и различными версиями Linux, хотя, как показывает наш далеко не полный обзор, этими ОС не исчерпывается набор альтернатив. Уменьшение количества предложений на рынке коммерческих ОС (как и на рынке, например, микропроцессоров) является, по-видимому, объективной тенденцией, следствием жестокой конкурентной борьбы. Такое уменьшение, а в перспективе – монопольный захват рынка одной системой, имеет для пользователей свои положительные и отрицательные стороны. С одной стороны, при отсутствии разнообразия значительно снижаются затраты пользователей на интеграцию разных технологий в своей информационной системе. С другой, пользователь рискует попасть в полную зависимость от одного производителя. В такой зависимости уже сейчас находятся пользователи ПЭВМ.

Мы не склонны возлагать большие надежды на законодательные меры, призванные ограничить монополию на рынке ОС, однако, не склонны и преувеличивать опасность возможной монополизации. Как показывает практика развития информационных технологий, такие

тенденции не новы, и они никогда не могли реализоваться до конца. Спектр задач, решаемых вычислительной техникой, постоянно расширяется (в настоящее время к таким принципиально новым задачам можно отнести мобильные и встроенные вычисления, обработку мультимедийной информации в реальном времени, добычу данных, супервычисления), и монополисты просто не могут (и не хотят) своевременно удовлетворять растущие потребности пользователя. В такой ситуации пользователей могут спасти альтернативные решения – как сохранившиеся в отдельных "экологических нишах" коммерческой сферы, так и некоммерческие решения. Такой изначально некоммерческой альтернативой была в свое время ОС Unix, в такой роли сейчас выступает Linux.

При внедрении новых решений в информационных технологиях очень часто оказывается востребованным опыт ранних решений, оставшийся в стороне от основного пути развития или подавленный конкурентами. Поэтому нам представляется чрезвычайно важным обеспечение сохранения такого опыта и его творческого использования.

Список литературы

1. Брукс П.Ф. Мифический человеко-месяц или как создаются программные системы. – СПб.:Символ-Плюс, 2001.
2. Брюзгин А.А. Программирование в системе Windows. – М.:Малип, 1992.
3. Деревянко А.С. Системное программное обеспечение персональных ЭВМ. – Харьков:ХГПУ, 1994.
4. Касперски К. Атака на Windows NT. // "Журнал сетевых решений LAN", 2000, №12.

5. Кинг А. Windows 95 изнутри. – С-Пб:Питер, 1995.
6. Крэнц Дж., Майзелл Э., Уилльямз Р. Операционная система OS/2. Возможности, функции и приложения. – М.:Мир, 1991.
7. Кузьминский М. Z-архитектура. // "Открытые системы", 2001, №10 (<http://www.osp.ru/os/2001/10/010.htm>).
8. Кузьминский М. Эволюция подсистемы ввода-вывода мэйнфреймов IBM. // "Открытые системы", 1999, №1 (<http://www.osp.ru/os/1999/01/25.htm>).
9. Максвелл С. Ядро Linux в комментариях. – К.:ДиаСофт, 2000.
10. Ресурсы Windows NT. – С-Пб.: BVH–Санкт-Петербург, 1995.
11. Солтис Ф. Основы AS/400. – М.: Изд.отд. "Русская редакция" ТОО "Channel Trading Ltd." – 1998.
12. AIX 5L Version 5.1 Technical Reference: Kernel and Subsystems, v1, v2. – http://publib.boulder.ibm.com/cgi-bin/ds_rslt#1
13. **AMX RTOS and Related Manuals.** – <http://www.kadak.com/html/kdkp1010.htm>
14. Bach M.J. Design of the Unix Operating System. – Prentice-Hall Inc, Englewood, 1986. (Русский перевод есть, например, в <http://lib.ru/BACH/>).
15. Boyce J. Inside Windows 98. – Macmillan Computer Publishing, 1999.
16. Caldera Product Support Documentation – <http://www.caldera.com/support/docs/>
17. Consumer and Embedded Technologies – http://java.sun.com/products/OV_embeddedProduct.html
18. Free BSD Handbook. – http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/index.html
19. Gosling J., Joy B., Steele G., Bracha G. The Java Language Specification. Second Edition. – Sun Microsystems Inc., 2000.

20. Hallberg B., Katy I. OS/2 Certification Handbook. – New Riders Publishing, Indianapolis, 1995.
21. Hoskins J. IBM System/390. A Business Perspective. – Maximum Press, NJ, 1997.
22. HP-UX 11i – information library. – <http://www.hp.com/products1/unix/operating/hpux11i/infolibrary/index.html>
23. iSeries Library – <http://www-1.ibm.com/servers/eserver/iseries/library/>
24. Joung J. Exploring IBM's New Age Mainframes. – Maximum Press, NJ, 1995.
25. Lindholm T., Yellin F. Java™ Virtual Machine Specification. Second Edition. – Sun Microsystems Inc., 2000.
26. Linux Documentation Project. – <http://www.linux.org/docs/ldp/index.html>
27. Loepere K. Mach 3 Kernel Principles. – <http://www-2.cs.cmu.edu/afs/cs/project/mach/public/www/doc/osf.html>
28. MacOS 8 and 9 Developer Documentation. – <http://developer.apple.com/techpubs/macos8/>
29. MacOS X Developer Documentation. – <http://developer.apple.com/techpubs/macosx/>
30. Open Unix 8 Documentation. – <http://docsrv.caldera.com:1997/>
31. Palm OS: Developer Documentation. – <http://www.palmos.com/dev/tech/docs/>
32. QNX System Architecture. – http://www.qnx.com/literature/qnx_sysarch/
33. Santo B. Embedded Batle Royale, – IEEE Spectrum, August 2001.
34. Solaris 8 Software Developer Collection. – <http://docs.sun.com/>
35. Solomon D.A., Russinovich M. Inside Microsoft Windows 2000. – Microsoft Press, 2000

- 36. The Be Book. – http://www.be.com/documentation/be_book/
- 37. Tru64 Unix version 5.1A Online Documentation. – http://www.tru64unix.compaq.com/docs/pub_page/V51A_DOCS/V51A_DOCLIST.HTM
- 38. VSE/ESA Internet Library – <http://www-1.ibm.com/servers/eserver/zseries/os/vse/library/library.htm>
- 39. Windows CE. Product Documentation. – <http://msdn.microsoft.com/library/default.asp?url=/nhp/default.asp?contentid=28000437>
- 40. z/Architecture. Principles of Operation, IBM SA22-7832-00
- 41. z/OS Internet Library – <http://www-1.ibm.com/servers/eserver/zseries/zos/bkserv/>
- 42. z/VM Internet Library – <http://www.vm.ibm.com/library/>