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 практическими 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 - 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 Motorolla 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 (160х160 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;

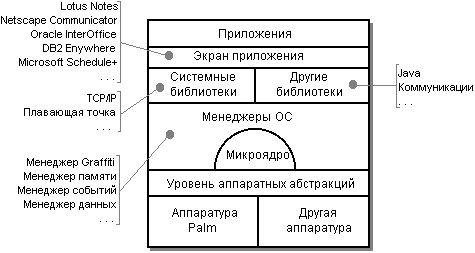
ОС и приложения функционируют в условиях очень ограниченного объема ресурсов, прежде всего - памяти;

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.



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 Kadak. 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 запраштвает 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 нереентерабельных 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 C 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.

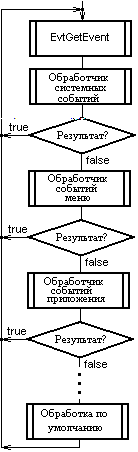


Рисунок 7.2 Обработка событий в приложении PalmOS

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 выполняютcя. 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.

Управление памятью и данными

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 неперемещаемую 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 MemHandeLock 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 неперемещаемых - в конце кучи.

Каждая порция памяти имеет свой локальный (в пределах данной карты памяти) идентификатор. Для неперемещаемой 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 память для ее расширения. 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. Основной системный вызов 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 OS 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 пикселов (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 иконки 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 cтандартный 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.

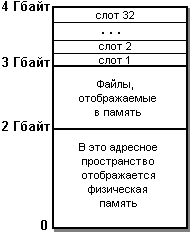


Рисунок 7.3 Виртуальное адресное пространство Windows CE

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 значению 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. Последующие операции 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.

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 Intel+Windows - 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 PowerMac, также выпускаемых фирмой Motorolla. В настоящее время все новые модели Macintosh базируются на PowerMac.

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 [28] нам не удалось найти определения архитектуры Mac OS, мы возьмем на себя смелость определить ее как модульно-иерархическую, как представлено на рисунке 8.1.

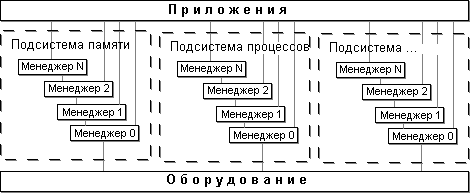


Рисунок 8.1 Архитектура Mac OS

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 их 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 Майт 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 68К, 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 68K. 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 удалаямый. Помеченный таким образом сегментный блок может быть удален из памяти при нехватке памяти.

Архитектура CFM

В новых версиях Mac OS на M68K и PowerMac 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 застежку, но для связывания с подключением (plug-in) создается отдельная застежка 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 застежке. Если для кодового ответвления создается несколько соединений, то все соединения совместно используют один экземпляр кода. Для ответвления данных возможена 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". Обрабатывая 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 связи 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 Mas OS работает на двух разных аппаратных платформах, имеются некоторые различия в форматах структур данных для платформ M68K и PowerMac (например, в PowerMac 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 поддерживается 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).

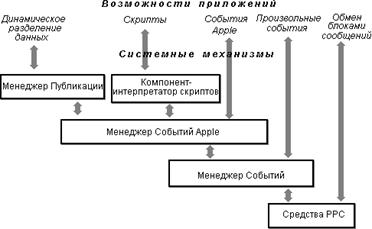


Рисунок 8.2 Средства взаимодействия приложений в Mac OS

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 AppleSkript, и, следовательно, выполнять действия, определенные в скрипте. При посылке сообщения такому приложению интерпретатор использует специальный ресурс типа "aete" (Apple event terminology extention), 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, неперемещаемая 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 инициируют 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 B-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 экстентами 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 экстентах файлов, не поместившихся в основные записи файлов в каталоге. Этот файл общий для всего тома и также организован как B-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 B-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, дружественность 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.

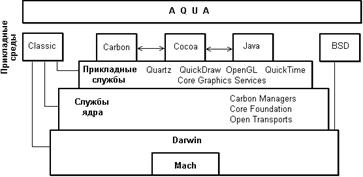
Набор программных модулей, которые обеспечивают базовые функции работы с интерфейсной графикой (рисование, перемещение, вращение и т.п. образов), называется в Mac OS QuickDraw. Приложения используют QuickDraw неявным образом, когда обращаются к другим менеджерам графического интерфейса для создания интерфейсных 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 невытесняющей 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.



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 CarnegieMellon (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 (нитями);

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 создаtтся 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 ISSO 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 (собственно Quartz) и базовых графических служб или сервера окон (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 8.2) 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, диалоги, 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 128х128 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 замечательным 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 фирма предлагает компьютеры iMac, iBook, PowerBook, Server G4 на процессорах PowerMac 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, претендующим на обеспечение совместимости.

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 BeBox 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 BeOSв 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 С - 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 Потоки 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 выделяются 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 разделения времени, приоритеты со значениями 100 и выше - класс приоритетов реального времени.

Приоритеты разделения 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 разделения 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 разделения 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, знающая идентификатор нити-корреспондента, может записать в этот буфер сообщение системным вызовом send\_data(), а нить - владелец буфера выбирает сообщение системным вызовом recive\_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 (системный вызов create\_port()) задается его емкость - число сообщений, которое может сохраняться в порте. Попытка писать в переполненный порт или читать из пустого порта, естественно, приводит к блокировке нити. Однако, есть варианты системных вызовов (write\_port\_etс() и read\_port\_etc()), которые к блокировке не приводят. Но 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 границе 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 экстентов, каждый экстент - целое число последовательных блоков. План размещения файла представляет собой массив описателей экстентов. 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 находится 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.

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.



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 активным 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 (proxy).

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

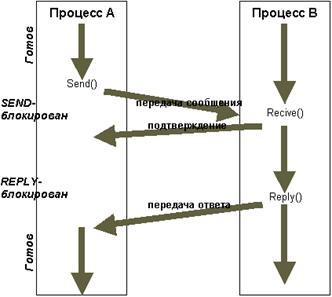


Рисунок 10.2 Сценарий взаимодействия процессов при посылке сообщения

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.

RECIVE-блокированный процесс - запросил прием сообщения и ожидает его поступления. На рисунке 10.2 состояние RECIVE-блокировки не показано, в него мог бы попасть Процесс B, если бы выполнил системный вызов Recive() прежде, чем Процесс A 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 A) передать сообщение и готовности другого процесса (Процесса B) 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 RECIVE-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 -Crecive() - позволяет процессу проверить наличие сообщений для него и, таким образом, избежать RECIVE-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, но 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.

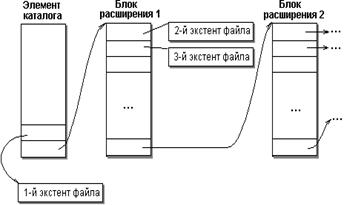


Рисунок 10.4 Размещение файла на диске QNX

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

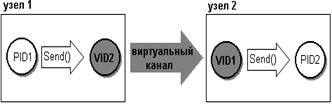


Рисунок 10.5 Посылка сообщения через виртуальный канал

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 графической системы 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 встроенного 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.

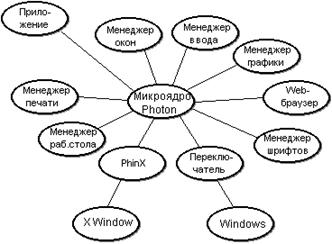


Рисунок 10.6 Архитектура графической системы Photon

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 Photon, 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 в этот ряд. Каждая область имеет две маски для проходящих через нее событий: 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.

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 бит, 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-технологию, 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.

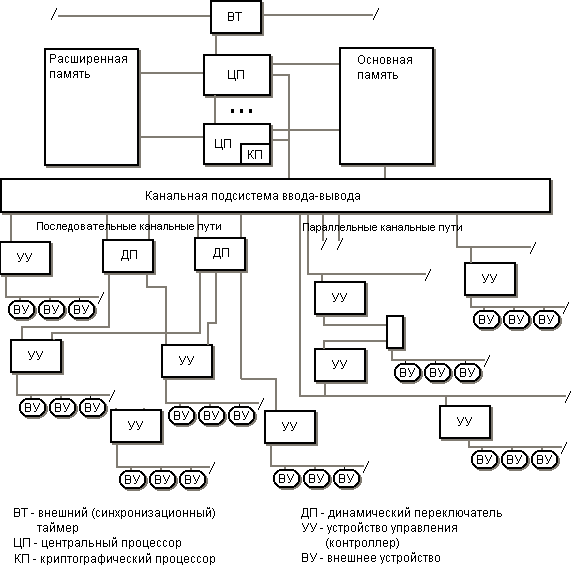


Рисунок 12.1 Логическая структура z-сервера

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 ключей защиты 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. Системные (привилегированные) программы выполняются с нулевым ключом защиты, что дает им доступ к любой странице памяти.

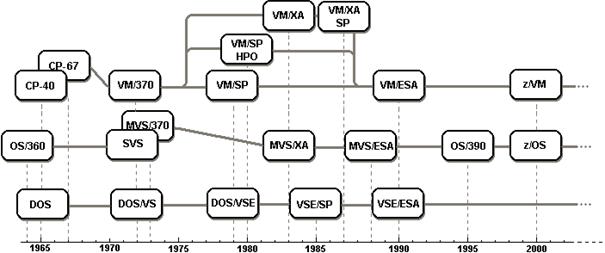
Система ввода-вывода основывается на каналах ввода-вывода, описанных нами в главе 6 части I. Однако там мы описали строго иерархическое подключение "канал - контроллер - устройство", которое применялось в ранних реализациях. Современная архитектура мейнфреймов обеспечивает более сложную схему подключений 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 канальные пути ESCON и FICON состоят из двух фибероптических кабелей, динамических переключателей и контроллеров. Динамическое переключение может быть выполнено между двумя любыми последовательными канальными путями в этой же или в другой канальной 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.



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.

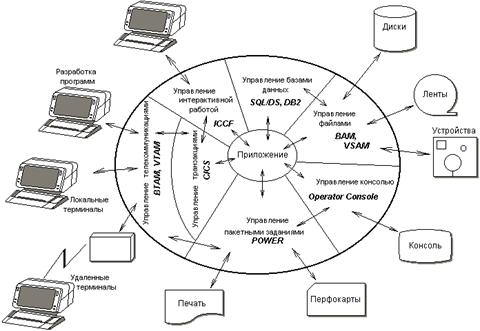


Рисунок 12.3 Среда выполнения приложения в VSE/ESA

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.

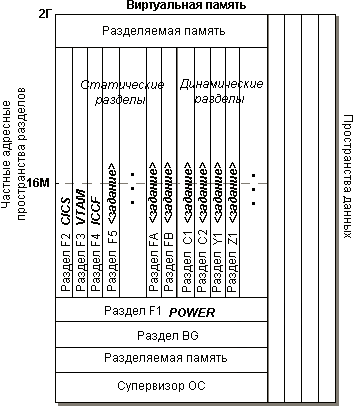


Рисунок 12.4 Структура памяти VSE/ESA

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 разделения 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 задания 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 перекладывают 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. BAM-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 BAM, создаваемые на диске DOSRES при инсталляции системы:

системная библиотека IJSYSRS.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 B+-дерево. Кроме того, "листья" дерева связаны в линейный список, что позволяет осуществлять быстрый последовательный доступ к данным файла. VSAM поддерживает физические стуктуры 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 BAM, 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 BAM 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.

ICCF обеспечивает собственные библиотеки и подбиблиотеки, предназначенные прежде всего для хранения текстов программ и заданий. Файлы в библиотеках ICCF 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 ICCF 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 ICCF использует компонент VSE/CICS (Customer Information Control System), который обязательно должен устанавливаться вместе с ICCF.

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 разноплатформенных 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/BTAM (Basic Telecommunication Access Method) является базовым телекоммуникационным методом доступа, обеспечивающим управление локальными и удаленными устройствами с использованием протоколов BSC или Asynch. BTAM 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.

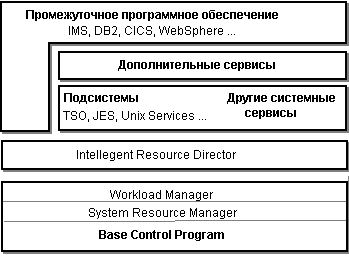


Рисунок 12.5 Структура программного обеспечения z/OS

Мы отметили, что развитие этой ОС происходило исключительно эволюционным путем. Внедрение новых возможностей управления ресурсами в ОС происходит, как правило, по следующему сценарию:

новый управляющий сервис разрабатывается и внедряется как отдельный программный продукт, продаваемый отдельно от ОС;

новый программный продукт включается в комплект поставки ОС;

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-битого 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.

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 в 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 ECB 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 ECB, 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 ECB 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 CHAP. 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 ECB, используемых для синхронизации выполнения задачи и ее подзадач. Однако блоки ECB являются более универсальным средством синхронизации выполнения. ECB 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'а.

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 параллелелизмом - 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 POWER 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 получат 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 кадры 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 кадров. 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 мейнйфреймы 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 настройке, 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'а) 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'а, 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'а), 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 канальные 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 простой для пользователя установки программного обеспечения) и z/OS Wizards - web-базированные 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'а - 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 (ВМ). ВМ в z/VM является аналогом процесса в других ОС: это тот "субъект", которому CP выделяет ресурсы. ВМ 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 ВМ CP. PSW ВМ определяет для выполняющейся на ВМ программы состояние "супервизор" (привилегированное состояние). PSW же реального оборудования при выполнении такой программы определяет состояние "задача" (непривилегированное). При попытке программы, выполняющейся на ВМ, выполнить привилегированную команду происходит исключение, и управление получает CP. CP распознает причину исключения и выполняет для ВМ привилегированную команду или моделирует ее выполнение, после чего возвращает управление ВМ. Исключение и его обработка скрыты от ВМ, ВМ кажется, что ее привилегированная команда выполнилась на реальном оборудовании.

СP на выбор моделирует для ВМ архитектуры нескольких поколений мейнфреймов - от 370/XA до z900, а также виртуальную архитектуру ESA/XC (eXtended Confuguration), в которой ВМ могут быть доступны (при авторизации) адресные пространства других ВМ. В число компонентов архитектуры ВМ 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:

ОС, специально созданные как гостевые, которые могут работать только в среде ВМ под управлением гипервизора - CMS и GCS.

Полнофункциональные другие ОС мейнфреймов (VSE, z/OS и ее предшественники, Linux for zSeries), выполняющиеся "не зная" о существовании гипервизора.

Те же ОС, но адаптированные для выполнения в среде ВМ, адаптация состоит в том, что исключается дублирование функций в гипервизоре и супервизоре.

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 CP и т.д.

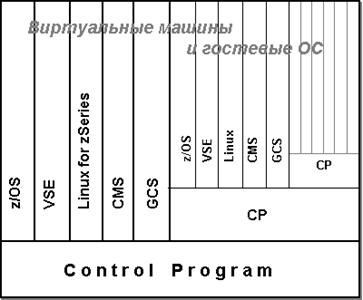


Рисунок 12.7 CP и виртуальные машины

Определение ВМ является квазипостоянным: оно создается один раз, а затем используется многократно. Определение ВМ сохраняется в каталоге CP, основным содержанием элемента каталога CP является описание ресурсов, выделяемых ВМ. При запуске ВМ на выполнение CP 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 CP. Все VMDBLK связаны в кольцевой список.

Управление памятью

Возможно, главным ресурсом, которым управляет CP, является реальная память, и с этой точки зрения CP может создавать ВМ трех типов:

Тип V=V - ВМ, которой выделяется только виртуальная память, требуемый размер памяти дя ВМ обеспечивается за счет динамической трансляции адресов и страничного свопинга.

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.

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 CP 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 VM, поэтому эти типы ВМ создаются только при наличии действительной необходимости в них (например, для задач реального времени).

Если под управлением CP работают только ВМ типа V=V, то ядро CP 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 CP 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 CP, а еще выше - динамическая страничная область.

Динамическая страничная область содержит:

управляющие блоки СP;

нерезидентные модули CP;

блоки управления памятью CP;

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 CP 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. CP 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 сисок периодически переупорядочивается 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.

CP 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. СP 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, CP 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, CP 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 CP), 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 CP. Однако, и в случае закрепления устройство для ВМ является виртуальным. Его адрес в ВМ не совпадает с реальным адресом устройства, CP 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 CP 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 диковом накопителе. Описание минидисков, принадлежащих ВМ (адрес на реальном диске и размер), хранится в каталоге CP. 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 CP, а управляющие воздействия для него интерпретируются CP с использованием буферов в памяти и программных кодов.

CMS

Диалоговая управляющая система СMS (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 CP. 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 пользователями, 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 POSIX, 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 является поддержка архитектуры SNA как части z/VM без помощи какой-либо другой ОС. Эта поддержка выполняется продуктом ACF/VTAM (Advanced Communication Functiom/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, независящий 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 большим):

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 ВМ. 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 CHAP. 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.

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;

аплеты - программы, выполняемые в среде Web-броузера, поддерживающего платформу Java (Sun HotJava, Netscape Communicator, Microsoft Internet Explorer), такие программы могут передаваться по Internet и выполняться на компьютере клиента;

сервлеты и корпоративные бины - Java-программы, серверные компоненты распределенных приложений;

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 С++ 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.

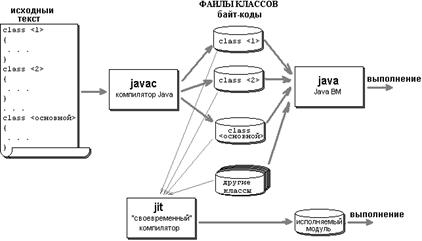


Рисунок 13.1 Выполнение приложения в платформе Java

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.

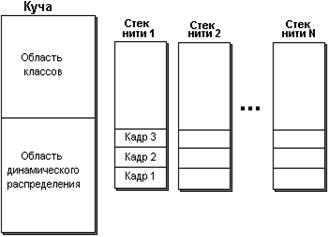


Рисунок 13.2 Основные области памяти Java VM

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 optop;

структуры среды выполнения, на которую ссылается регистр frame.

Эти области показаны на рисунке 13.3.

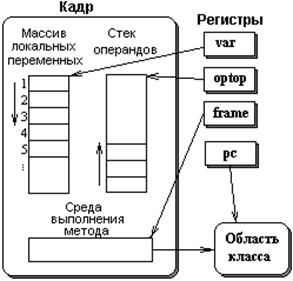


Рисунок 13.3 Структура и связи кадра

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 кадр 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 происходят 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 VM 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, представляющих 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 аплета, 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.

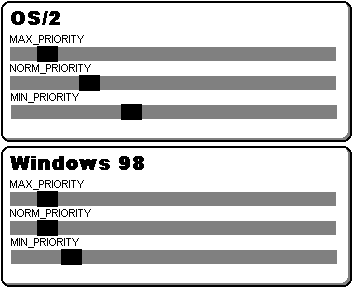


Рисунок 13.5 "Гонки" в разных операционных средах (движение справа налево).

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 начинает 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;

yeld() - 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 умолчанию 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 выполнение 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 "потоковыми 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 MAJC 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 A на первый процессор, а метод B - на второй для предположительного вычисления. Поскольку B - 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 MAJC 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 освобождается, а следовательно, и освобождаются и все локальные переменные. 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 становиться 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 случает 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 непредказуем, 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 обстоятельством с точки зрения защиты.

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 С/С++ 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, обрабатывающийся 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, соответствующего 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:

проверяется, 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.

Аплеты являются наиболее критическими с точки зрения безопасности Java-программами, поскольку аплет загружается из Internet, возможно, из непроверенного источника. Естественно, недопустимым является предоставление программе, пришедшей "неизвестно откуда" доступа к ресурсам локального компьютера. Поэтому для аплетов введены весьма жесткие ограничения на выполнение. Аплету 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 аплета;

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 способам. 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-не



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 (опционные) 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 состоят в том, что и JVM, и библиотека классов являются конфигурируемыми, то есть, 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.

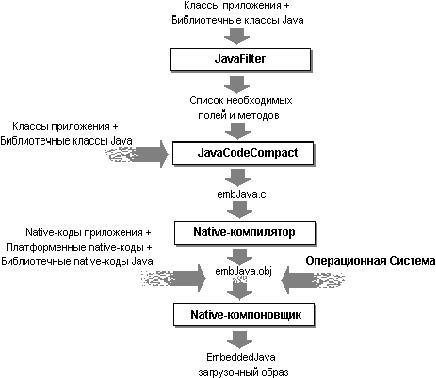


Рисунок 13.7 Процесс разработки приложения EmbeddedJava

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, с которым Java сейчас интегрируется (стандарт Java Standard Extension for XML).

Первый успех технологий Java был обеспечен прежде всего аплетами, 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.