**МОНОЛИТНЫЕ СИСТЕМЫ**

Несомненно, такая организация операционной системы является самой распростра- ненной. Здесь вся операционная система работает как единая программа в режиме ядра. Операционная система написана в виде набора процедур, связанных вместе в одну большую исполняемую программу. При использовании этой технологии каж- дая процедура может свободно вызвать любую другую процедуру, если та выполняет какое-нибудь полезное действие, в котором нуждается первая процедура. Возможность вызвать любую нужную процедуру приводит к весьма высокой эффективности работы системы, но наличие нескольких тысяч процедур, которые могут вызывать друг друга сколь угодно часто, нередко делает ее громоздкой и непонятной. Кроме того, отказ в любой из этих процедур приведет к аварии всей операционной системы.

Для построения исполняемого файла монолитной системы необходимо сначала ском- пилировать все отдельные процедуры (или файлы, содержащие процедуры), а затем связать их вместе, воспользовавшись системным компоновщиком. Здесь, по существу, полностью отсутствует сокрытие деталей реализации — каждая процедура видна любой другой процедуре (в отличие от структуры, содержащей модули или пакеты, в которых основная часть информации скрыта внутри модулей и за пределами модуля его про- цедуры можно вызвать только через специально определяемые точки входа).

Тем не менее даже такие монолитные системы могут иметь некоторую структуру. Служ- бы (системные вызовы), предоставляемые операционной системой, запрашиваются путем помещения параметров в четко определенное место (например, в стек), а затем выполняется инструкция *trap*. Эта инструкция переключает машину из пользова- тельского режима в режим ядра и передает управление операционной системе (шаг *6* на рис. 1.17). Затем операционная система извлекает параметры и определяет, какой системный вызов должен быть выполнен. После этого она перемещается по индексу в таблице, которая в строке *k* содержит указатель на процедуру, выполняющую систем- ный вызов *k* (шаг *7* на рис. 1.17).

Такая организация предполагает следующую базовую структуру операционной си- стемы:

1. Основная программа, которая вызывает требуемую служебную процедуру.

2. Набор служебных процедур, выполняющих системные вызовы.

3. Набор вспомогательных процедур, содействующих работе служебных процедур. В этой модели для каждого системного вызова имеется одна ответственная за него слу-

жебная процедура, которая его и выполняет. Вспомогательные процедуры выполняют действия, необходимые нескольким служебным процедурам, в частности извлечение данных из пользовательских программ. Таким образом, процедуры делятся на три уровня (рис. 1.21).







**Рис. 1.21.** Простая структурированная модель монолитной системы

В дополнение к основной операционной системе, загружаемой во время запуска ком- пьютера, многие операционные системы поддерживают загружаемые расширения, в числе которых драйверы устройств ввода-вывода и файловые системы. Эти ком- поненты загружаются по мере надобности. В UNIX они называются библиотеками общего пользования. В Windows они называются **DLL-библиотеками** (Dynamic-Link Libraries — динамически подключаемые библиотеки). Они находятся в файлах с расши- рениями имен .dll, и в каталоге C:\Windows\system32 на системе Windows их более 1000.

**МНОГОУРОВНЕВЫЕ СИСТЕМЫ**

Обобщением подхода, показанного на рис. 1.21, является организация операционной системы в виде иерархии уровней, каждый из которых является надстройкой над ниже- лежащим уровнем. Первой системой, построенной таким образом, была система THE, созданная в Technische Hogeschool Eindhoven в Голландии Э. Дейкстрой (E. W. Dijkstra) и его студентами в 1968 году. Система THE была простой пакетной системой для голландского компьютера Electrologica X8, имевшего память 32 K 27-разрядных слов. Как показано в табл. 1.3, у системы было шесть уровней. Уровень 0 занимался рас- пределением ресурса процессора (процессорного времени), переключением между процессами при возникновении прерываний или истечении времени таймера. Над уровнем 0 система состояла из последовательных процессов, каждый из которых мог быть запрограммирован без учета того, что несколько процессов были запущены на одном процессоре. Иными словами, уровень 0 обеспечивал основу многозадачности центрального процессора.

**Таблица 1.3.** Структура операционной системы THE

|  |  |
| --- | --- |
| **Уровень** | **Функция** |
| 5 | Оператор |
| 4 | Программы пользователя |
| 3 | Управление вводом-выводом |
| 2 | Связь оператора с процессом |
| 1 | Управление основной памятью и магнитным барабаном |
| 0 | Распределение ресурсов процессора и обеспечение многозадачного режима |

Уровень 1 управлял памятью. Он выделял процессам пространство в основной памяти и на магнитном барабане емкостью 512 К слов, который использовался для хранения частей процесса (страниц), не умещавшихся в оперативной памяти. На уровнях выше первого процессы не должны были беспокоиться о том, где именно они находятся, в памяти или на барабане. Программное обеспечение уровня 1 обеспечивает помещение страниц в память в то время, когда они необходимы, и удаление их из памяти, когда они не нужны.

Уровень 2 управлял связью каждого процесса с консолью оператора (то есть с пользо- вателем). Над этим уровнем каждый процесс фактически имел собственную консоль оператора. Уровень 3 управлял устройствами ввода-вывода и буферизацией инфор- мационных потоков в обоих направлениях. Над третьим уровнем каждый процесс мог работать с абстрактными устройствами ввода-вывода, имеющими определенные свойства. На уровне 4 работали пользовательские программы, которым не надо было заботиться о процессах, памяти, консоли или управлении вводом-выводом. Процесс системного оператора размещался на уровне 5.

Дальнейшее обобщение многоуровневой концепции было сделано в системе MULTICS. Вместо уровней для описания MULTICS использовались серии концентрических ко- лец, где внутренние кольца обладали более высокими привилегиями по отношению к внешним (что, собственно, не меняло сути многоуровневой системы). Когда процеду- ре из внешнего кольца требовалось вызвать процедуру внутреннего кольца, ей нужно было создать эквивалент системного вызова, то есть выполнить инструкцию *TRAP*, параметры которой тщательно проверялись на допустимость перед тем, как разрешить продолжение вызова. Хотя вся операционная система в MULTICS являлась частью адресного пространства каждого пользовательского процесса, аппаратура позволяла определять отдельные процедуры (а фактически — сегменты памяти) как защищенные от чтения, записи или выполнения.

Следует отметить, что система уровней в конструкции THE играла лишь вспомогатель- ную роль, поскольку все части системы в конечном счете компоновались в единую ис- полняемую программу, а в MULTICS кольцеобразный механизм существовал главным образом в процессе выполнения и реализовывался за счет аппаратного обеспечения.

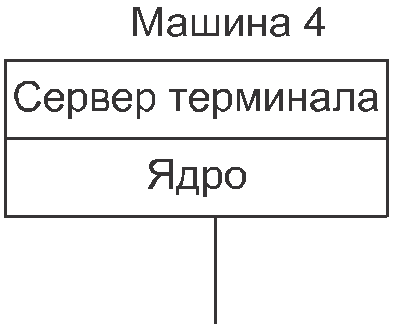
Преимущества кольцеобразного механизма проявлялись в том, что он мог быть легко расширен и на структуру пользовательских подсистем. Например, профессор может написать программу для тестирования и оценки студенческих программ и запустить ее в кольце *n*, а студенческие программы будут выполняться в кольце *n* + 1, так что студенты не смогут изменить свои оценки.

**КЛИЕНТ-СЕРВЕРНАЯ МОДЕЛЬ**

Небольшая вариация идеи микроядер выражается в обособлении двух классов про- цессов: **серверов**, каждый из которых предоставляет какую-нибудь службу, и **клиентов**, которые пользуются этими службами. Эта модель известна как **клиент-серверная**. До- вольно часто самый нижний уровень представлен микроядром, но это не обязательно. Суть заключается в наличии клиентских процессов и серверных процессов.

Связь между клиентами и серверами часто организуется с помощью передачи со- общений. Чтобы воспользоваться службой, клиентский процесс составляет сообще- ние, в котором говорится, что именно ему нужно, и отправляет его соответствующей службе. Затем служба выполняет определенную работу и отправляет обратно ответ. Если клиент и сервер запущены на одной и той же машине, то можно провести опре- деленную оптимизацию, но концептуально здесь речь идет о передаче сообщений.

Очевидным развитием этой идеи будет запуск клиентов и серверов на разных ком- пьютерах, соединенных локальной или глобальной сетью (рис. 1.23). Поскольку кли- енты связываются с серверами путем отправки сообщений, им не обязательно знать, будут ли эти сообщения обработаны локально, на их собственных машинах, или же они будут отправлены по сети на серверы, расположенные на удаленных машинах. Что касается интересов клиента, следует отметить, что в обоих случаях происходит одно и то же:



**Рис. 1.23.** Клиент-серверная модель, реализованная с помощью сети

отправляются запросы и возвращаются ответы. Таким образом, клиент- серверная модель является абстракцией, которая может быть использована как для отдельно взятой машины, так и для машин, объединенных в сеть.

Становится все больше и больше систем, привлекающих пользователей, сидящих за домашними компьютерами, в качестве клиентов, а большие машины, работающие где-нибудь в другом месте, — в качестве серверов. Фактически по этой схеме работает большая часть Интернета. Персональные компьютеры отправляют запросы на полу- чение веб-страницы на сервер, и эта веб-страница им возвращается. Это типичная картина использования клиент-серверной модели при работе в сети.

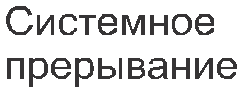
**ВИРТУАЛЬНЫЕ МАШИНЫ**

Первые выпуски OS/360 были системами исключительно пакетной обработки. Но многие пользователи машин IBM/360 хотели получить возможность интерактивной работы с использованием терминала, поэтому различные группы разработчиков как в самой корпорации IBM, так и за ее пределами решили написать для этой машины системы с разделением времени. Позже была выпущена официальная система раз деления времени — TSS/360, и когда она наконец-то дошла до потребителей, то была настолько громоздкой и медлительной, что под нее было переоборудовано всего лишь несколько вычислительных центров. В конечном счете от этого проекта отказались, после того как на него уже было потрачено 50 млн долларов (Graham, 1970).

**VM/370**

Группа из научного центра IBM Scientific Center в Кембридже (Массачусетс) раз- работала совершенно другую систему, которую IBM в конечном итоге приняла как законченный продукт. Эта система, первоначально называвшаяся CP/CMS, а позже переименованная в VM/370 (Seawright and MacKinnon, 1979), была основана на сле- дующем проницательном наблюдении: система с разделением времени обеспечивает, во-первых, многозадачность, а во-вторых, расширенную машину с более удобным интерфейсом, чем у простого оборудования. Сущность VM/370 заключается в полном разделении этих двух функций.

Основа системы, известная как **монитор виртуальных машин**, запускается непосред- ственно на обычном оборудовании и обеспечивает многозадачность, предоставляя верхнему уровню не одну, а несколько виртуальных машин (рис. 1.24). Но, в отличие от всех других операционных систем, эти виртуальные машины не являются машинами с расширенной архитектурой. Они не





**Рис. 1.24.** Структура VM/370 с тремя запущенными системами CMS

поддерживают файлы и другие полезные свой- ства. Вместо этого они являются точной копией исходной аппаратуры, включающей режим ядра и пользователя, устройства ввода-вывода, прерывания и все остальное, что есть у настоящей машины.

Поскольку каждая виртуальная машина идентична настоящему оборудованию, на каж- дой из них способна работать любая операционная система, которая может быть запу- щена непосредственно на самом оборудовании. На разных виртуальных машинах могут быть запущены разные операционные системы, как это часто и происходит на самом деле. Изначально на системах VM/370 пользователи запускали в своих виртуальных машинах OS/360 или одну из других больших операционных систем пакетной обработ- ки или обработки транзакций, в то время как другие запускали однопользовательскую интерактивную систему **CMS** (Conversational Monitor System — система диалоговой обработки ) для пользователей системы разделения времени.

Когда программа под управлением операционной системы CMS выполняет системный вызов, он перехватывается в системное прерывание операционной системы на своей собственной виртуальной машине, а не на VM/370, как это было бы при ее запуске на реальной, а не на виртуальной машине. Затем CMS выдает обычные команды ввода- вывода для чтения своего виртуального диска или другие команды, которые могут ей понадобиться для выполнения этого вызова. Эти команды ввода-вывода перехватыва- ются VM/370, которая выполняет их в рамках моделирования реального оборудования. При полном разделении функций многозадачности и предоставления машины с рас- ширенной архитектурой каждая из составляющих может быть намного проще, гибче и удобнее для обслуживания.

В своем современном перерождении z/VM обычно используется для запуска несколь- ких полноценных операционных систем, а не упрощенных, однопользовательских си- стем вроде CMS. Например, на машинах zSeries можно запустить одну или несколько виртуальных машин Linux, а наряду с ними — обычные операционные системы IBM.

**Повторное открытие виртуальных машин**

Хотя в IBM виртуальные машины используются уже четыре десятилетия и ряд других компаний, включая Oracle и Hewlett-Packard, недавно добавили поддержку вирту- альных машин к своим высокопроизводительным промышленным серверам, тем не менее, по большому счету, идея виртуализации в мире персональных компьютеров до последнего времени практически игнорировалась. Но сейчас сочетание новых по- требностей, нового программного обеспечения и новых технологий придало этой теме особую актуальность.

Сначала о потребностях. Многие компании традиционно запускали свои почтовые сер- веры, веб-серверы, FTP-серверы и все остальные серверы на отдельных компьютерах, иногда имеющих различные операционные системы. Виртуализация рассматривается ими как способ запуска всех этих серверов на одной и той же машине с возможностью избежать при этом отказа всех серверов при отказе одного из них.

Виртуализация популярна также в мире веб-хостинга. Без нее клиенты вынуждены выбирать между **общим хостингом**1 (который дает им только учетную запись на веб- сервере, но не позволяет управлять его программным обеспечением) и выделенным хостингом (который предоставляет им их собственную, очень гибкую, но не оправ- дывающую затрат машину при небольших или средних по объему веб-сайтах). Когда компания, предоставляющая услуги веб-хостинга (хостинг-провайдер), сдает в аренду виртуальные машины, на одной физической машине может быть запущено множе- ство виртуальных машин, каждая из которых превращается в полноценную машину. Клиенты, арендовавшие виртуальную машину, могут запускать на ней какие угодно операционную систему и программное обеспечение, но за часть стоимости выделен- ного сервера (поскольку та же самая физическая машина одновременно поддерживает множество виртуальных машин).

Другой вариант использования виртуализации предназначен для конечных пользо- вателей, которым необходима возможность одновременного запуска двух или более операционных систем, например Windows и Linux, поскольку некоторые из любимых ими приложений работают под управлением только одной из этих операционных си- стем. Такая ситуация показана на рис. 1.25, *а*, при этом термин «монитор виртуальной машины» заменен на «**гипервизор первого типа**» (type 1 hypervisor), который широко используется в наши дни из-за краткости при наборе по сравнению с первым вариан- том. Но для многих авторов они являются взаимозаменяемыми.

Excel Word Mplayer Apollon

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | |  |  | |
|  | | |  | | | ... |
| Windows | | | Linux | | |
|  | | | | | | |

Процесс гостевой ОС Процесс гостевой ОС

й

|  |  |  |  |
| --- | --- | --- | --- |
|  | Процесс | | |
|  | | основно  ОС | |
| Гостевая ОС | |  |  |
| Модель машины | |  | |
| Основная операционная система | | | |

дуль а

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | | | |
|  | | Мо ядр | | |
| Гостевая ОС | |
|  | |  |
|  | |  | |  |
| Основная операционная система | | |  | |

в

**Рис. 1.25.** Гипервизор: *а* — тип 1; *б* — чистый гипервизор, тип 2;

*в* — практический гипервизор, тип 2 (перевод)

Привлекательность виртуальных машин сомнениям не подвергалась, проблема заклю- чалась в их реализации. Чтобы запустить на компьютере программное обеспечение вир- туальных машин, его центральный процессор должен быть готов к работе в этом режиме (Popek and Goldberg, 1974). Проблема заключается в следующем. Когда операционная система, запущенная на виртуальной машине (в режиме пользователя), выполняет при- вилегированные инструкции, например изменение слова состояния программы — PSW или операцию ввода-вывода, необходимо, чтобы оборудование осуществило перехват данных инструкций и вызов монитора виртуальных машин, который выполнит их про- граммную эмуляцию. На некоторых центральных процессорах, особенно на Pentium, его предшественниках и их клонах, попытки выполнения привилегированных инструкций в режиме пользователя просто игнорируются. Эта особенность исключает создание виртуальных машин на таком оборудовании, чем объясняется недостаточный интерес к ним в мире x86. Конечно, существовали интерпретаторы для Pentium, такие как Bochs, которые запускались на этом процессоре, но при потере производительности обычно в один-два порядка они не подходили для серьезной работы.

В 1990-х годах и в первые годы нового тысячелетия был реализован ряд научно-ис- следовательских проектов, в частности Disco в Стэнфорде (Bugnion et al., 1997) и Xen в Кембридже (Barham et al., 2003). Эти исследования привели к появлению нескольких коммерческих продуктов (например, VMware Workstation и Xen), и интерес к вир- туальным машинам снова вырос. Сегодня в число популярных гипервизоров кроме VMware иXen входят KVM (для ядра Linux), VirtualBox (от Oracle) и Hyper-V (от Microsoft).

Некоторые из этих ранних исследовательских проектов улучшили производитель- ность по сравнению с интерпретаторами типа Bochs путем трансляции блоков кода на лету, сохранения их во внутреннем кэше и повторного использования результата трансляции в случае их нового исполнения. Это существенно повысило производи- тельность и привело к созданию того, что сейчас называется **моделями машин** (machine simulators) (рис. 1.25, *б*). Но хотя эта технология, известная как **двоичная трансляция** (binary translation), помогла улучшить ситуацию, получившиеся системы, несмотря на то что они неплохо подходили для публикаций на академических конференциях, по-прежнему не отличались быстротой для использования в коммерческих средах, где производительность имеет весьма большое значение.

Следующим шагом в улучшении производительности стало добавление модуля ядра (рис. 1.25, *в*) для выполнения ряда трудоемких задач. Сложившаяся сейчас прак- тика показывает, что все коммерчески доступные гипервизоры, такие как VMware Workstation, используют эту гибридную стратегию (а также имеют множество других усовершенствований). Все их называют гипервизорами типа 2, поэтому и мы (хотя и не вполне охотно) последуем этой тенденции и будем использовать данное назва- ние далее в этой книге, даже при том, что лучше бы было назвать их гипервизорами

типа 1.7, чтобы отразить тот факт, что они не являются в полном смысле програм- мами пользовательского режима. В главе 7 будет дано подробное описание того, как работает VMware Workstation и чем занимаются его компоненты.

На практике действительным различием между гипервизорами типа 1 и типа 2 явля- ется то, что в типе 2 для создания процессов, сохранения файлов и т. д. используется **основная операционная система** (host operating system) и ее файловая система. Ги- первизор типа 1 не имеет основной поддержки и должен выполнять все эти функции самостоятельно.

После запуска гипервизор типа 2 считывает установочный компакт-диск (или файл образа компакт-диска) для выбора **гостевой операционной системы** (guest operation system) и установки гостевой ОС на виртуальный диск, который является просто большим файлом в файловой системе основной операционной системы. Гипервизор типа 1 этого делать не может по причине отсутствия основной операционной системы, в которой можно было бы хранить файлы.

Во время своей загрузки гостевая операционная система делает все то же самое, что и на настоящем оборудовании, как обычно запуская некоторые фоновые процессы, а за- тем графический пользовательский интерфейс. С точки зрения пользователя, гостевая операционная система ведет себя точно так же, как при запуске непосредственно на оборудовании, хотя в данном случае ситуация совсем иная.

Другим подходом к обработке управляющих инструкций является модификация операционной системы с целью их удаления. Этот подход не является настоящей виртуализацией, он относится к **паравиртуализации**. Более подробно виртуализация будет рассмотрена в главе 7.

**Виртуальная машина Java**

Виртуальные машины используются, правда, несколько иным образом и в другой об- ласти — для запуска программ на языке Java. Когда компания Sun Microsystems изо- брела язык программирования Java, она также изобрела и виртуальную машину (то есть архитектуру компьютера), названную **JVM** (Java Virtual Machine — виртуальная машина Java). Компилятор Java создает код для JVM, который затем обычно выполня- ется программным интерпретатором JVM. Преимущество такого подхода состоит в том, что код для JVM может доставляться через Интернет на любой компьютер, имеющий JVM-интерпретатор, и запускаться на этом компьютере. Если бы компилятор создавал двоичные программы, например для SPARC или x86, их нельзя было бы так же легко куда угодно доставлять и где угодно запускать. (Разумеется, Sun могла бы создать ком- пилятор, производящий двоичные файлы для SPARC, а затем распространить SPARC- интерпретатор, но у JVM намного более простая для интерпретации архитектура.) Дру- гим преимуществом использования JVM является то, что при правильной реализации интерпретатора, что не является такой уж простой задачей, полученные JVM-программы могут быть проверены с точки зрения безопасности, а затем выполнены в защищенной среде, не имея возможности похитить данные или нанести любой другой вред

**. ЭКЗОЯДРА**

Вместо клонирования настоящей машины, как это делается в виртуальных машинах, существует иная стратегия, которая заключается в их разделении, иными словами, в предоставлении каждому пользователю подмножества ресурсов. При этом одна виртуальная машина может получить дисковые блоки от 0 до 1023, другая — блоки от

1024 до 2047 и т. д.

Самый нижний уровень, работающий в режиме ядра, — это программа под названием **экзоядро** (Engler et al., 1995). Ее задача состоит в распределении ресурсов между вир- туальными машинами и отслеживании попыток их использования, чтобы ни одна из машин не пыталась использовать чужие ресурсы. Каждая виртуальная машина может запускать собственную операционную систему, как на VM/370 и на Pentium в режиме виртуальных машин 8086, с тем отличием, что каждая машина ограничена использо- ванием тех ресурсов, которые она запросила и которые были ей предоставлены.

Преимущество схемы экзоядра заключается в том, что она исключает уровень ото- бражения. При других методах работы каждая виртуальная машина считает, что она имеет собственный диск с нумерацией блоков от 0 до некоторого максимума. Поэтому монитор виртуальных машин должен вести таблицы преобразования адресов на диске (и всех других ресурсов). При использовании экзоядра необходимость в таком пере- назначении отпадает. Экзоядру нужно лишь отслеживать, какой виртуальной машине какие ресурсы были переданы. Такой подход имеет еще одно преимущество — он отде- ляет многозадачность (в экзоядре) от пользовательской операционной системы (в про- странстве пользователя) с меньшими затратами, так как для этого ему необходимо всего лишь не допускать вмешательства одной виртуальной машины в работу другой.