Глава 3
Управление памятью

Память представляет собой очень важный ресурс, требующий четкого управления. Несмотря на то что в наши дни объем памяти среднего домашнего компьютера в десятки тысяч раз превышает ресурсы IBM 7094, бывшего в начале 60-х годов самым большим компьютером в мире, размер компьютерных программ растет быстрее, чем объем памяти. Закон Паркинсона можно перефразировать следующим образом: «Программы увеличиваются в размерах, стремясь заполнить всю память, доступную для их размещения». В этой главе мы рассмотрим, как операционные системы создают из памяти абстракции и как они этими абстракциями управляют.

В идеале каждому программисту хотелось бы иметь предоставленную только ему неограниченную по объему и скорости работы память, которая к тому же не теряет своего содержимого при отключении питания. Раз уж мы так размечтались, то почему бы не сделать память еще и совсем дешевой? К сожалению, существующие технологии пока не могут дать нам желаемое. Может быть, способ создания такой памяти удастся изобрести именно вам.

Тогда чем же нам придется довольствоваться? Со временем была разработана концепция **иерархии памяти**, согласно которой компьютеры обладают несколькими мегабайтами очень быстродействующей, дорогой и энергозависимой кэш-памяти, несколькими гигабайтами памяти, средней как по скорости, так и по цене, а также несколькими терабайтами памяти на довольно медленных, сравнительно дешевых дисковых накопителях, не говоря уже о сменных накопителях, таких как DVD-диски и флеш-устройства USB. Превратить эту иерархию в абстракцию, то есть в удобную модель, а затем управлять этой абстракцией — и есть задача операционной системы.

Та часть операционной системы, которая управляет иерархией памяти (или ее частью), называется **менеджером**, или **диспетчером**, **памяти**. Он предназначен для действенного управления памятью и должен следить за тем, какие части памяти используются, выделять память процессам, которые в ней нуждаются, и освобождать память, когда процессы завершат свою работу.

В этой главе будут рассмотрены несколько разных схем управления памятью, начиная с очень простых и заканчивая весьма изощренными. Поскольку управление кэш-памятью самого нижнего уровня обычно осуществляется на аппаратном уровне, основное внимание будет уделено программистской модели оперативной памяти и способам эффективного управления ее использованием. Все, что касается абстракций, создаваемых для энергонезависимого запоминающего устройства — диска, и управления этими абстракциями, будет темой следующей главы. Начнем с истоков, и в первую очередь рассмотрим самую простую из возможных схем, а затем постепенно будем переходить к изучению все более сложных.

3.1. Память без использования абстракций

Простейшей абстракцией памяти можно считать полное отсутствие какой-либо абстракции. Ранние универсальные машины (до 1960 года), ранние мини-компьютеры (до 1970 года) и ранние персональные компьютеры (до 1980 года) не использовали абстракции памяти. Каждая программа просто видела физическую память. Когда программа выполняла следующую команду

```
MOV REGISTER1,1000
```

компьютер просто перемещал содержимое физической ячейки памяти 1000 в *REGISTER1*. Таким образом, модель памяти, предоставляемая программисту, была простой физической памятью, набором адресов от 0 до некоторого максимального значения, где каждый адрес соответствовал ячейке, содержащей какое-нибудь количество бит, которое обычно равнялось восьми.

При таких условиях содержание в памяти сразу двух работающих программ не представлялось возможным. Если первая программа, к примеру, записывала новое значение в ячейку 2000, то она тем самым стирала то значение, которое сохранялось там второй программой. Работа становилась невозможной, и обе программы практически сразу же давали сбой.

Даже в условиях, когда в качестве модели памяти выступает сама физическая память, возможны несколько вариантов использования памяти. Три из них показаны на рис. 3.1. Операционная система может, как показано на рис. 3.1, *а*, размещаться в нижней части адресов, в оперативном запоминающем устройстве (ОЗУ), или, по-другому, в памяти с произвольным доступом — RAM (Random Access Memory), или она может размещаться в постоянном запоминающем устройстве (ПЗУ), или, по-другому, в ROM (Read-Only Memory), в верхних адресах памяти, как показано на рис. 3.1, *б*, или же драйверы устройств могут быть в верхних адресах памяти, в ПЗУ, а остальная часть системы — в ОЗУ, в самом низу, как показано на рис. 3.1, *в*. Первая модель прежде использовалась на универсальных машинах и мини-компьютерах, но на других машинах использовалась довольно редко. Вторая модель использовалась на некоторых КПК и на встроенных системах. Третья модель использовалась на ранних персональных компьютерах (например, на тех, которые работали под управлением MS-DOS), где часть системы, размещавшаяся в ПЗУ, называлась базовой системой ввода-вывода — **BIOS** (Basic Input Output System). Недостаток моделей *а* и *в* заключается в том, что ошибка в программе пользователя может затереть операционную систему и, возможно, с весьма пагубными последствиями (такими, как порча содержимого диска).

При таком устройстве системы памяти процессы, как правило, могут запускаться только по одному. Как только пользователь наберет команду, операционная система копирует запрошенную программу с диска в память, после чего выполняет эту программу. Когда процесс завершает свою работу, операционная система отображает символ приглашения и ожидает ввода новой команды. По получении команды она загружает в память новую программу, записывая ее поверх первой программы.

Единственный способ добиться хоть какой-то параллельности в системе, не имеющей абстракций памяти, — это запустить программу с несколькими потока-

ми. Поскольку предполагается, что все имеющиеся в процессе потоки видят один и тот же образ памяти, факт их принудительного применения проблемы не составляет. Хотя эта идея вполне работоспособна, она не нашла широкого применения, поскольку людям зачастую необходим одновременный запуск не связанных друг с другом программ, а этого абстракция потоков не обеспечивает. Более того, столь примитивные системы, не обеспечивающие абстракций памяти, вряд ли могут предоставить абстракцию потоков.

Рис. 3.1. Три простых способа организации памяти при наличии операционной системы и одного пользовательского процесса (существуют также и другие варианты)

Запуск нескольких программ без абстракций памяти

Тем не менее даже в отсутствие абстракций памяти одновременный запуск нескольких программ вполне возможен. Для этого операционная система должна сохранить все содержимое памяти в файле на диске, а затем загрузить и запустить следующую программу. Поскольку одновременно в памяти присутствует только одна программа, конфликтов не возникает. Эта концепция называется заменой данных (или свопингом) и будет рассмотрена чуть позже.

При наличии некоторого специального дополнительного оборудования появляется возможность параллельного запуска нескольких программ даже без использования свопинга. На ранних моделях IBM 360 эта проблема решалась следующим образом: память делилась на блоки по 2 Кбайта, каждому из которых присваивался 4-битный защитный ключ, содержащийся в специальных регистрах за пределами центрального процессора. Машине с объемом памяти в 1 Мбайт нужно было иметь лишь 512 таких 4-битных регистров, и все хранилище ключей занимало в итоге 256 байт памяти. Слово состояния программы — PSW (Program Status Word) также содержало 4-битный ключ. Аппаратное обеспечение IBM 360 перехватывало любую попытку запущенного процесса получить доступ к памяти с ключом защиты, отличающимся от ключа PSW. Поскольку изменить ключи защиты могла только операционная система, пользовательские процессы были защищены от вмешательства в работу друг друга и в работу самой операционной системы.

Тем не менее это решение имело серьезный недостаток, показанный на рис. 3.2. Здесь изображены две программы, каждая из которых имеет объем 16 Кбайт. Они

показаны на рис. 3.2, *а* и *б*. Первая из них закрашена, чтобы показать, что у нее другой ключ памяти, чем у второй. Первая программа начинается с перехода на ячейку памяти с адресом 24, в которой содержится команда *MOV*. Вторая программа начинается с перехода на ячейку памяти с адресом 28, в которой содержится команда *CMP*. Команды, не имеющие отношения к рассматриваемому вопросу, на рисунке не показаны. Когда две программы загружаются друг за другом в память, начиная с ячейки с адресом 0, мы получаем ситуацию, показанную на рис. 3.2, *в*. В этом примере мы предполагаем, что операционная система находится в верхних адресах памяти и поэтому не показана.

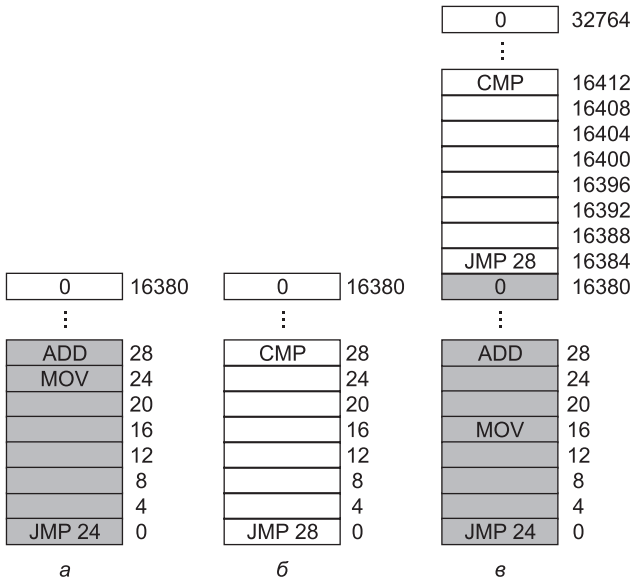

Рис. 3.2. Иллюстрация проблемы перемещения: программа объемом 16 Кбайт (*а*); еще одна программа объемом 16 Кбайт (*б*); две программы, последовательно загруженные в память (*в*)

После загрузки программы могут быть запущены на выполнение. Поскольку у них разные ключи памяти, ни одна из них не может навредить другой. Но проблема имеет иную природу. При запуске первой программы будет выполнена команда *JMP 24* и, как и ожидалось, осуществлен переход к другой команде. Эта программа будет успешно работать.

Но после того как первая программа проработает достаточно долго, операционная система может принять решение на запуск второй программы, которая была загружена над первой программой, начиная с адреса 16384. Первой исполняемой командой будет *JMP 28*, осуществляющая переход к команде *ADD* первой программы вместо того, чтобы перейти к предполагаемой команде *CMP*. Скорее всего, это приведет к сбою программы на первой же секунде.

В данном случае суть проблемы состоит в том, что обе программы ссылаются на абсолютный адрес физической памяти, что совершенно не соответствует нашим желаниям. Нам нужно чтобы каждая программа ссылалась на свой собственный,

занимаемый ею набор адресов. Давайте вкратце рассмотрим, как это достигается. На IBM 360, например, в процессе загрузки буквально на лету осуществлялась модификация второй программы, при этом использовалась технология, известная как статическое перемещение. Она работала следующим образом: когда программа загружалась с адреса 16384, в процессе загрузки к каждому адресу в программе прибавлялось постоянное значение 16384. При всей исправности работы этого механизма он был не самым универсальным решением, которое к тому же замедляло загрузку. Более того, это решение требовало дополнительной информации обо всех исполняемых программах, служащей признаком, в каких словах содержатся, а в каких не содержатся перемещаемые адреса. В результате чего «28» на рис. 3.2, *б* должно было подвергнуться перемещению, а инструкция вроде

```
MOV REGISTER1,28
```

которая помещает число 28 в *REGISTER1,* должна была остаться нетронутой. Нужен был какой-нибудь способ, позволяющий сообщить загрузчику, какое из чисел относится к адресу, а какое — к константе.

И наконец, как отмечалось в главе 1, в компьютерном мире истории свойственно повторяться. Хотя прямая адресация физической памяти, к сожалению, осталась в прошлом, в устаревших устройствах памяти универсальных компьютеров, миникомпьютеров, настольных компьютеров и ноутбуков, дефицит абстракций памяти — вполне обычное явление во встроенных системах и смарт-картах. В наши дни устройства вроде радиоприемников, стиральных машин и микроволновых печей заполнены программным обеспечением (в ПЗУ), и в большинстве случаев их программы используют адресацию к абсолютной памяти. Все это неплохо работает, поскольку все программы известны заранее, и пользователи не могут запускать на бытовых устройствах какие-нибудь собственные программы.

Сложные операционные системы присутствуют только в высокотехнологичных встроенных устройствах (например, в сотовых телефонах), а более простые устройства их не имеют. В некоторых случаях у них тоже есть операционная система, но это всего лишь библиотека, связанная с прикладной программой, которая предоставляет системные вызовы для выполнения операций ввода-вывода и других общих задач. Простым примером популярной операционной системы, представляющей собой библиотеку, может послужить **e-cos**.

3.2. Абстракция памяти: адресные пространства

В конечном счете предоставление физической памяти процессам имеет ряд серьезных недостатков. Во-первых, если пользовательские программы могут обращаться к каждому байту памяти, они легко могут преднамеренно или случайно испортить операционную систему, раздробить ее код и довести до остановки работы (в отсутствие специального аппаратного обеспечения наподобие ключевых схем и блокировок на IBM 360). Эта проблема присутствует даже при запуске всего лишь одной пользовательской программы (приложения). Во-вторых, при использовании этой

модели довольно сложно организовать одновременную (поочередную, если имеется лишь один центральный процессор) работу нескольких программ. На персональных компьютерах вполне естественно наличие нескольких одновременно открытых программ (текстовый процессор, программа электронной почты, веб-браузер), с одной из которых в данный момент взаимодействует пользователь, а работа других возобновляется щелчком мыши. Так как этого трудно достичь при отсутствии абстракций на основе физической памяти, нужно что-то предпринимать.

3.2.1. Понятие адресного пространства

Чтобы допустить одновременное размещение в памяти нескольких приложений без создания взаимных помех, нужно решить две проблемы, относящиеся к защите и перемещению. Примитивное решение первой из этих проблем мы уже рассматривали на примере IBM 360: участки памяти помечались защитным ключом, и ключ выполняемого процесса сличался с ключом каждого выбранного слова памяти. Но этот подход не решал второй проблемы, хотя она могла быть решена путем перемещения программ в процессе их загрузки, но это было слишком медленным и сложным решением.

Более подходящее решение — придумать для памяти новую абстракцию: адресное пространство. Так же как понятие процесса создает своеобразный абстрактный центральный процессор для запуска программ, понятие адресного пространства создает своеобразную абстрактную память, в которой существуют программы. **Адресное пространство** — это набор адресов, который может быть использован процессом для обращения к памяти. У каждого процесса имеется свое собственное адресное пространство, независимое от того адресного пространства, которое принадлежит другим процессам (за исключением тех особых обстоятельств, при которых процессам требуется совместное использование их адресных пространств).

Понятие адресного пространства имеет весьма универсальный характер и появляется во множестве контекстов. Возьмем телефонные номера. В США и многих других странах местный телефонный номер состоит обычно из семизначного номера. Поэтому адресное пространство телефонных номеров простирается от 0000000 до 9999999, хотя некоторые номера, к примеру, те, что начинаются с 000, не используются. С ростом количества сотовых телефонов, модемов и факсов это пространство стало слишком тесным, а в этом случае необходимо использовать больше цифр. Адресное пространство портов ввода-вывода процессора Pentium простирается от 0 до 16 383. Протокол IPv4 обращается к 32-разрядным номерам, поэтому его адресное пространство простирается от 0 до $2^{32} - 1$ (опять-таки с некоторым количеством зарезервированных номеров).

Адресное пространство не обязательно должно быть числовым. Набор интернет-доменов .com также является адресным пространством. Это адресное пространство состоит из всех строк длиной от 2 до 63 символов, которые могут быть составлены из букв, цифр и дефисов, за которыми следует название домена — .com. Теперь вам должна стать понятной сама идея, в которой нет ничего сложного.

Немного сложнее понять, как каждой программе можно выделить свое собственное адресное пространство, поскольку адрес 28 в одной программе означает

иное физическое место, чем адрес 28 в другой программе. Далее мы рассмотрим простой способ, который ранее был распространен, но вышел из употребления с появлением возможностей размещения на современных центральных процессорах более сложных (и более совершенных) схем.

Базовый и ограничительный регистры

В простом решении используется весьма примитивная версия **динамического перераспределения памяти**. При этом адресное пространство каждого процесса просто проецируется на различные части физической памяти. Классическое решение, примененное на машинах от CDC 6600 (первого в мире суперкомпьютера) до Intel 8088 (сердца первой модели IBM PC), заключается в оснащении каждого центрального процессора двумя специальными аппаратными регистрами, которые обычно называются **базовым** и **ограничительным** регистрами. При использовании этих регистров программы загружаются в последовательно расположенные свободные области памяти без модификации адресов в процессе загрузки, как показано на рис. 3.2, *в*. При запуске процесса в базовый регистр загружается физический адрес, с которого начинается размещение программы в памяти, а в ограничительный регистр загружается длина программы. На рис. 3.2, *в* при запуске первой программы базовыми и ограничительными значениями, загружаемыми в эти аппаратные регистры, будут соответственно 0 и 16 384. При запуске второй программы будут соответственно использованы значения 16 384 и 32 768. Если непосредственно над второй будет загружена и запущена третья программа, имеющая объем 16 Кбайт, значениями базового и ограничительного регистров будут 32 768 и 16 384.

При каждой ссылке процесса на память с целью извлечения команды или записи слова данных аппаратура центрального процессора перед выставлением адреса на шине памяти добавляет к адресу, сгенерированному процессом, значение базового регистра. Одновременно аппаратура проверяет, не равен ли предлагаемый адрес значению ограничительного регистра или не превышает ли он это значение, и в этом случае генерируется отказ и доступ прерывается. Если взять первую команду второй программы, показанной на рис. 3.2, *в*, то процесс выполняет команду

```
JMP 28
```

но аппаратура рассматривает ее как команду

```
JMP 16412
```

поэтому переход осуществляется, как и ожидалось, на команду *CMP*. Значения базовых и ограничительных регистров при выполнении второй программы на рис. 3.2, *в* показаны на рис. 3.3.

Использование базового и ограничительного регистров — это простой способ предоставления каждому процессу своего собственного закрытого адресного пространства, поскольку к каждому автоматически сгенерированному адресу перед обращением к памяти добавляется значение базового регистра. Многие реализации предусматривают такую защиту базового и ограничительного регистров, при которой изменить их значения может только операционная система. Именно так был устроен компьютер CDC 6600 в отличие от компьютеров на основе Intel 8088, у которых не было даже ограничительного регистра. Но у последних было не-

сколько базовых регистров, позволяющих, к примеру, реализовать независимое перемещение текста и данных программы, но не предлагающих какой-либо защиты от ссылок за пределы выделенной памяти.

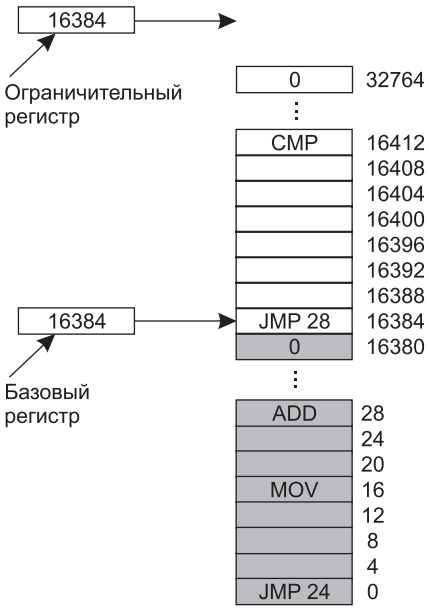

Рис. 3.3. Для предоставления каждому процессу отдельного адресного пространства могут использоваться базовый и ограничительный регистры

Недостатком перемещений с использованием базовых и ограничительных регистров является необходимость применения операций сложения и сравнения к каждой ссылке на ячейку памяти. Сравнение может осуществляться довольно быстро, но сложение является слишком медленной операцией из-за затрат времени на вспомогательный сигнал переноса, если, конечно, не используются специальные сумматоры.

3.2.2. Свопинг

Если у компьютера достаточный объем памяти для размещения всех процессов, то все рассмотренные до сих пор схемы будут в той или иной степени работоспособны. Но на практике суммарный объем оперативной памяти, необходимый для размещения всех процессов, зачастую значительно превышает имеющийся объем ОЗУ. На обычных Windows- или Linux-системах при запуске компьютера могут быть запущены около 40–60 или более процессов. Например, при установке приложения Windows зачастую выдаются команды, чтобы при последующих запусках системы запускался процесс, единственной задачей которого была бы проверка наличия обновлений для этого приложения. Такой процесс запросто может занять 5–10 Мб памяти. Остальные фоновые процессы проверяют наличие входящей

почты, входящих сетевых подключений и многое другое. И все это еще до того, как будет запущена первая пользовательская программа. Современные солидные пользовательские прикладные программы могут запросто занимать при своей работе от 50 до 200 и более Мбайт памяти. Следовательно, постоянное содержание всех процессов в памяти требует огромных объемов и не может быть осуществлено при дефиците памяти.

С годами для преодоления перегрузки памяти были выработаны два основных подхода. Самый простой из них, называемый **свопингом**, заключается в размещении в памяти всего процесса целиком, в запуске его на некоторое время, а затем в сбросе его на диск. Бездействующие процессы большую часть времени хранятся на диске и в нерабочем состоянии не занимают пространство оперативной памяти (Хотя некоторые из них периодически активизируются, чтобы проделать свою работу, после чего опять приостанавливаются). Второй подход называется **виртуальной памятью**, он позволяет программам запускаться даже в том случае, если они находятся в оперативной памяти лишь частично. Далее в этом разделе мы рассмотрим свопинг, а в последующих разделах будет рассмотрена и виртуальная память.

Работа системы с использованием свопинга показана на рис. 3.4. Изначально в памяти присутствует только процесс *A*. Затем создаются или появляются в памяти путем свопинга с диска процессы *B* и *C*. На рис. 3.4, *г* процесс *A* за счет свопинга выгружается на диск. Затем появляется процесс *D* и выгружается из памяти процесс *B*. И наконец, снова появляется в памяти процесс *A*. Поскольку теперь процесс *A* находится в другом месте, содержащиеся в нем адреса должны быть перестроены либо программным путем, при загрузке в процессе свопинга, либо (скорее всего) аппаратным путем в процессе выполнения программы. К примеру, для этого случая хорошо подойдут механизмы базового и ограничительного регистров.

Когда в результате свопинга в памяти создаются несколько свободных областей, их можно объединить в одну большую за счет перемещения при первой же возможности всех процессов в нижние адреса. Эта технология известна как **уплотнение памяти**. Но зачастую она не производится, поскольку отнимает довольно много процессорного времени. К примеру, на машине, оснащенной 1 Гбайт памяти, способной скопировать 4 байта за 20 нс, на уплотнение всего объема памяти может уйти около 5 с.

Стоит побеспокоиться о том, какой объем памяти нужно выделить процессу при его создании или загрузке в процессе свопинга. Если создаваемый процесс имеет вполне определенный неизменный объем, то выделение упрощается: операционная система предоставляет процессу строго необходимый объем памяти, ни больше ни меньше.

Но если сегмент данных процесса может разрастаться, к примеру, за счет динамического распределения памяти, как во многих языках программирования, то каждая попытка разрастания процесса вызывает проблему. Если к выделенному пространству памяти примыкает свободное место, то оно может быть распределено процессу, и он сможет расширяться в пределах этого свободного пространства. С другой стороны, если память, выделенная процессу, примыкает к памяти другого процесса, разрастающийся процесс должен быть либо перемещен на свободное пространство памяти, достаточное для его размещения, либо один или более процессов

должны быть сброшены на диск путем свопинга, чтобы образовалось достаточно свободного пространства. Если процесс не может разрастаться в памяти и область свопинга на диске уже заполнена, то процессу придется приостановить свою работу до тех пор, пока не освободится некоторое пространство памяти (или же этот процесс может быть уничтожен).

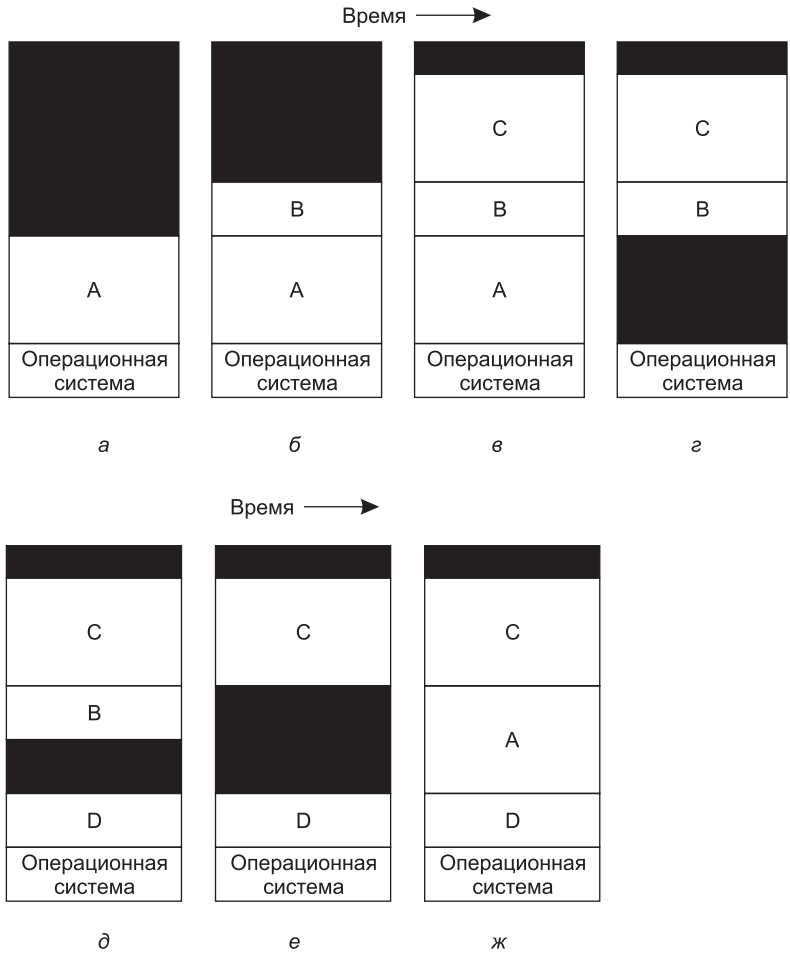

Рис. 3.4. Изменения в выделении памяти по мере появления процессов в памяти и выгрузки их из нее. Неиспользованные области памяти заштрихованы

Если предполагается, что большинство процессов по мере выполнения будут разрастаться, то, наверное, будет лучше распределять небольшой объем дополнительной памяти при каждой загрузке из области свопинга на диске в память или перемещении процесса, чтобы сократить потери, связанные со свопингом или перемещением процессов, которые больше не помещаются в отведенной им памяти. Но свопингу на диск должна подвергаться только реально задействованная память,