УПРАВЛЕНИЕ ПАМЯТЬЮ

В однозадачных системах основная память разделяется на две части:

одна часть - для операционной системы (резидентный монитор, ядро),

а вторая - для выполняющейся в текущий момент времени программы.

В многозадачных системах "пользовательская" часть памяти должна быть распределена для размещения нескольких процессов. Эта задача распределения выполняется операционной системой динамически и известна под названием управление памятью **(memory management)**.

Эффективное управление памятью жизненно важно для многозадачных систем. Если в памяти располагается только небольшое число процессов, то большую часть времен все эти процессы будут находиться в состоянии ожидания выполнения операций ввода-вывода, и загрузка процессора будет низкой.

Таким образом, желательно эффективное распределение памяти, позволяющее разместить в ней как можно больше процессов.

ОСНОВНЫЕ ТЕРМИНЫ, СВЯЗАННЫЕ С УПРАВЛЕНИЕМ ПАМЯТЬЮ

Кадр (frame) Блок основной памяти фиксированной длины

Страница (page) Блок данных фиксированной длины, находящийся во вторичной памяти (такой, как диск). Страница данных может быть временно скопирована в кадр основной памяти

Сегмент (segment) Блок данных переменной длины, находящийся во вторичной памяти. В доступную область основной памяти может быть скопирован сегмент полностью **(сегментация)**; возможно также разделение сегмента на страницы, которые

копируются в основную память по отдельности (комбинированная сегментация, или страничная организация памяти)

ТРЕБОВАНИЯ К УПРАВЛЕНИЮ ПАМЯТЬЮ

При рассмотрении различных механизмов и стратегий, связанных с управлением памятью, полезно помнить требования, которым они должны удовлетворять. Эти требования включают следующее.

- Перемещение
- Защита
- Совместное использование
- Логическая организация
- Физическая организация

Перемещение

В многозадачной системе доступная основная память в общем случае разделяется среди множества процессов. Обычно программист не знает заранее, какие программы будут резидентно находиться в основной памяти во время работы разрабатываемой им программы. Кроме того, для максимизации загрузки процессора желательно иметь большой пул процессов, готовых к выполнению, для чего требуется возможность загрузки и выгрузки активных процессов из основной памяти.

Требование, чтобы выгруженная из памяти программа была вновь загружена в то же самое место, где находилась и ранее, было бы слишком сильным ограничением.

Крайне желательно, чтобы программа могла быть перемещена ***(relocate)*** в другую область памяти.

Таким образом, заранее неизвестно, где именно будет размещена программа, а кроме того, программа может быть перемещена из одной области памяти в другую при свопинге.

Эти обстоятельства обусловливают наличие определенных технических требований к адресации, проиллюстрированных на рис. 7.1.

Рис. 7.1. Требования к адресации процесса

На рисунке представлен образ процесса. Для простоты предположим, что образ процесса занимает одну непрерывную область основной памяти.

Очевидно, что операционной системе необходимо знать местоположение управляющей информации процесса и стека выполнения, а также точки входа для начала выполнения процесса. Поскольку управлением памятью занимается операционная система и она же размещает процесс в основной памяти, соответствующие адреса она получает автоматически.

Однако, помимо получения операционной системой указанной информации, процесс должен иметь возможность обращаться к памяти в самой программе. Так, команды ветвления содержат адреса, указывающие на команды, которые должны быть выполнены после них; команды обращения к данным - адреса байтов или слов, с которыми они работают.

Так или иначе, но процессор и программное обеспечение операционной системы должны быть способны перевести ссылки в коде программы в реальные физические адреса, соответствующие текущему расположению программы в основной памяти.

Защита

Каждый процесс должен быть защищен от нежелательного воздействия других процессов, случайного или преднамеренного.

Следовательно, код других процессов не должен иметь возможности без разрешения обращаться к памяти данного процесса для чтения или записи. Однако удовлетворение требованию перемещаемости усложняет задачу защиты.

Поскольку расположение программы в основной памяти непредсказуемо, проверка абсолютных адресов во время компиляции невозможна.

Кроме того, в большинстве языков программирования возможно динамическое вычисление адресов во время выполнения (например, вычисление адреса элемента массива или указателя на поле структуры данных).

Следовательно, во время работы программы необходимо выполнять проверку всех обращений к памяти, генерируемых процессом, чтобы удостовериться, что все они - только к памяти, выделенной данному процессу.

К счастью, как вы увидите позже, механизмы поддержки перемещений обеспечивают и поддержку защиты.

Обычно пользовательский процесс не может получить доступ ни к какой части операционной системы - ни к коду, ни к данным. Код одного процесса не может выполнить команду ветвления, целевой код которой находится в другом процессе. Если не приняты специальные меры, код одного процесса не может получить доступ к данным другого процесса. Процессор должен быть способен прервать выполнение таких команд.

Заметим, что требования защиты памяти должны быть удовлетворены на уровне процессора (аппаратного обеспечения), а не на уровне операционной системы (программного обеспечения), поскольку операционная система не в состоянии предвидеть все обращения к памяти, которые будут выполнены программой.

Даже если бы такое было возможно, сканирование каждой программы в поиске предлагаемых нарушений защиты было бы слишком расточительным с точки зрения использования процессорного времени.

Следовательно, соответствующие возможности аппаратного обеспечения – единственное средство определения допустимости обращения к памяти (данным или коду) во время работы программы.

Совместное использование

Любой механизм защиты должен иметь достаточную гибкость для того, чтобы обеспечить возможность нескольким процессам обращаться к одной и той же области основной памяти.

Например, если несколько процессов выполняют один и тот же машинный код, то будет выгодно позволить каждому процессу работать с одной и той же копией этого кода, а не создавать собственную.

Процессам, сотрудничающим в работе над некоторой задачей, может потребоваться совместный доступ к одним и тем же структурам данных. Система управления памятью должна, таким образом, обеспечивать управляемый доступ к разделяемым областям памяти, при этом никоим образом не ослабляя защиту памяти. Как мы увидим позже, механизмы поддержки перемещений обеспечивают и поддержку совместного использования памяти.

Логическая организация

Практически всегда основная память в компьютерной системе организована как линейное (одномерное) адресное пространство, состоящее из последовательности байтов или слов.

Аналогично организована и вторичная память на своем физическом уровне.

Хотя такая организация и отражает особенности используемого аппаратного обеспечения, она не соответствует способу, которым обычно создаются программы.

Большинство программ организованы в виде модулей, одни из которых неизменны (только для чтения, только для выполнения), а другие содержат данные, которые могут быть изменены.

Если операционная система и аппаратное обеспечение компьютера могут эффективно работать с пользовательскими программами и данными, представленными модулями, то это обеспечивает ряд преимуществ.

1. Модули могут быть созданы и скомпилированы независимо один от другого, при этом все ссылки из одного модуля во второй разрешаются системой во время работы программы.

2. Разные модули могут получить разные степени защиты (только для чтения, только для выполнения) за счет весьма умеренных накладных расходов.

3. Возможно применение механизма, обеспечивающего совместное использование модулей разными процессами. Основное достоинство обеспечения совместного использования на уровне модулей заключается в том, что они соответствуют взгляду программиста на задачу и, следовательно, ему проще определить, требуется ли совместное использование того или иного модуля.

Инструментом, наилучшим образом удовлетворяющим данным требованиям, является сегментация, которая будет рассмотрена позже.

Физическая организация

Память компьютера разделяется как минимум на два уровня: основная и вторичная.

Основная память обеспечивает быстрый доступ по относительно высокой цене; кроме того, она энергозависима, т.е. не обеспечивает долговременное хранение.

Вторичная память медленнее и дешевле основной и обычно энергонезависима. Следовательно, вторичная память большой емкости может служить для долговременного хранения программ и данных, а основная память меньшей емкости – для хранения программ и данных, использующихся в текущий момент.

В такой двухуровневой структуре основной заботой системы становится организация потоков информации между основной и вторичной памятью.

Ответственность за эти потоки может быть возложена и на отдельного программиста, но это непрактично и нежелательно по следующим причинам.

1. Основной памяти может быть недостаточно для программы и ее данных. В этом случае программист вынужден прибегнуть к практике, известной как структуры с перекрытием - ***оверлеи (overlay),*** когда программа и данные организованы таким образом, что различные модули могут быть назначены одной и той же области памяти; основная программа при этом ответственна за перезагрузку модулей при необходимости. Даже при помощи соответствующего инструментария компиляции оверлеев разработка таких программ приводит к дополнительным затратам времени программиста.

2. Во многозадачной среде программист при разработке программы не знает, какой объем памяти будет доступен программе и где эта память будет располагаться.

Таким образом, очевидно, что задача перемещения информации между двумя уровнями памяти должна возлагаться на операционную систему. Эта задача является сущностью управления памятью.

РАСПРЕДЕЛЕНИЕ ПАМЯТИ

Главной операцией управления памятью является размещение программы в основной памяти для ее выполнения процессором. Практически во всех современных многозадачных системах эта задача предполагает использование сложной схемы, известной как виртуальная память.

Виртуальная память, в свою очередь, основана на использовании одной или обеих базовых технологий - *сегментации (segmentation)* и *страничной организации памяти (paging).*

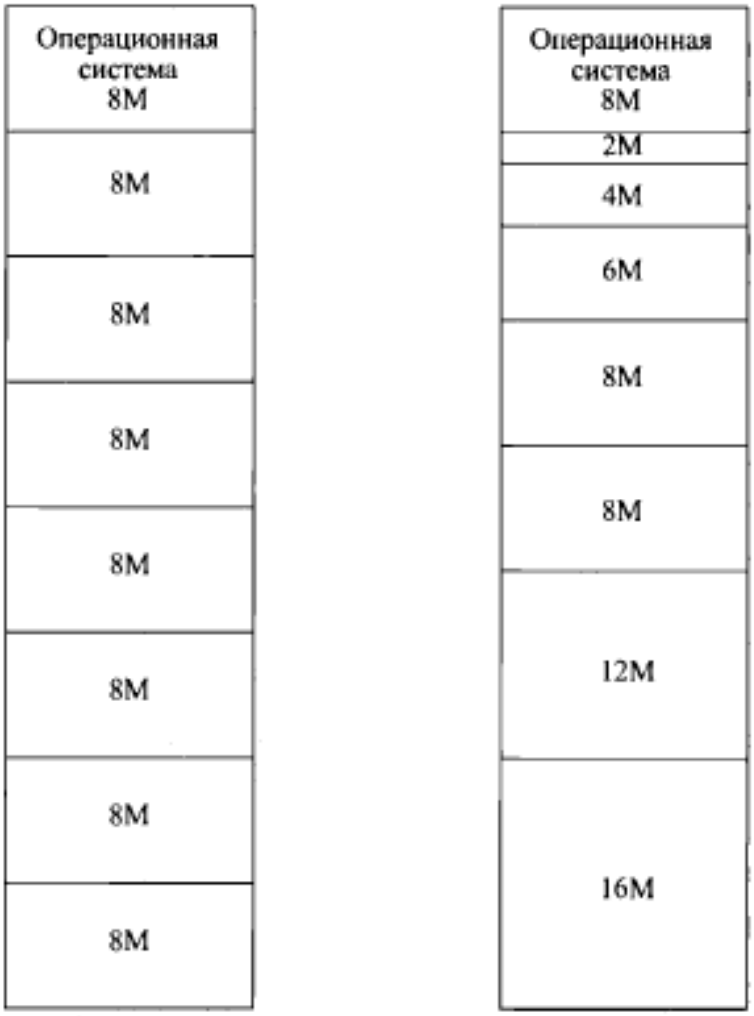
Таблица 7.2. Технологии управления памятью

Технология	Описание	Сильные стороны	Слабые стороны
Фиксированное распределение	Основная память разделяется на ряд статических разделов во время генерации системы. Процесс может быть загружен в раздел равного или большего размера	Простота реализации, малые системные накладные расходы	Неэффективное использование памяти из-за внутренней фрагментации, фиксированное максимальное количество активных процессов
Динамическое распределение	Разделы создаются динамически; каждый процесс загружается в раздел строго необходимого размера	Отсутствует внутренняя фрагментация, более эффективное использование основной памяти	Неэффективное использование процессора из-за необходимости уплотнения для противодействия внешней фрагментации
Простая страничная организация	Основная память разделена на ряд кадров равного размера. Каждый процесс разделен на некоторое количество страниц равного размера и той же длины, что и кадры. Процесс загружается путем загрузки всех его страниц в доступные, но не обязательно последовательные кадры	Отсутствует внешняя фрагментация	Наличие небольшой внутренней фрагментации

Простая сегментация	Каждый процесс распределен на ряд сегментов. Процесс загружается путем загрузки всех своих сегментов в динамические (не обязательно смежные) разделы	Отсутствует внутренняя фрагментация; по сравнению с динамическим распределением повышенная эффективность использования памяти и сниженные накладные расходы	Внешняя фрагментация
Страничная организация виртуальной памяти	Все, как при простой страничной организации, с тем исключением, что не требуется одновременно загружать все страницы процесса. Необходимые нерезидентные страницы автоматически загружаются в память	Нет внешней фрагментации; более высокая степень многозадачности; большое виртуальное адресное пространство	Накладные расходы из-за сложности системы управления памятью
Сегментация виртуальной памяти	Все, как при простой сегментации, с тем исключением, что не требуется одновременно загружать все сегменты процесса. Необходимые нерезидентные сегменты автоматически загружаются в память	Нет внутренней фрагментации; более высокая степень многозадачности; большое виртуальное адресное пространство; поддержка защиты и совместного использования	Накладные расходы из-за сложности системы управления памятью

Фиксированное распределение

В большинстве схем управления памятью мы будем полагать, что операционная система занимает некоторую фиксированную часть основной памяти и что остальная часть основной памяти доступна для использования многочисленным процессам. Простейшая схема управления этой доступной памятью - ее распределение на области с фиксированными границами.



а) Разделы одинакового размера б) Разделы разных размеров

Рис. 7.2. Пример фиксированного распределения 64-мегабайтовой памяти

Размеры разделов

Одна возможность состоит в использовании разделов одинакового размера. В этом случае любой процесс, размер которого не превышает размер раздела, может быть загружен в любой доступный раздел. Если все разделы заняты и нет ни одного процесса в состоянии готовности или работы, операционная система может выгрузить процесс из любого раздела и загрузить другой процесс, обеспечивая тем самым процессор работой.

При использовании разделов с одинаковым размером имеются две трудности.

- Программа может быть слишком велика для размещения в разделе. В этом случае программист должен разрабатывать программу, использующую оверлеи, с тем чтобы в любой

момент времени ей требовался только один раздел основной памяти. Когда требуется модуль, который в настоящий момент отсутствует в основной памяти, пользовательская программа должна сама загрузить этот модуль в раздел памяти программы (независимо от того, является ли этот модуль кодом или данными).

• Использование основной памяти при этом крайне неэффективно. Любая программа, независимо от ее размера, занимает раздел целиком. Так, в нашем примере программа размером менее мегабайта все равно будет занимать целиком раздел в 8 Мбайт; при этом остаются неиспользованными 7 Мбайт блока. Этот феномен появления неиспользованной памяти из-за того, что загружаемый блок по размеру меньше раздела, называется ***внутренней фрагментацией (intemal fragmentation).***

Бороться с этими трудностями (хотя и не устранить полностью) можно посредством использования разделов разных размеров.

Алгоритм размещения

В том случае, когда разделы имеют одинаковый размер, размещение процессов в памяти представляет собой тривиальную задачу. Не имеет значения, в каком из свободных разделов будет размещен процесс. Если все разделы заняты процессами, которые не готовы к немедленной работе, любой из них может быть выгружен для освобождения памяти для нового процесса.

Принятие решения о том, какой именно процесс следует выгрузить, - задача планировщика.

Когда разделы имеют разные размеры, есть два возможных подхода к назначению процессов разделам памяти. Простейший путь состоит в том, чтобы каждый процесс размещался в наименьшем разделе, способном полностью вместить данный процесс.

Рис. 7.3. Распределение памяти при фиксированном распределении

В таком случае для каждого раздела требуется очередь планировщика, в которой хранятся выгруженные из памяти процессы, предназначенные для данного раздела памяти (рис. 7.3, а). Преимущество такого подхода заключается в том, что процессы могут быть распределены между разделами памяти так, чтобы минимизировать внутреннюю фрагментацию.

Хотя этот метод представляется оптимальным с точки зрения отдельного раздела, он не оптимален с точки зрения системы в целом.

Представим, что в системе, изображенной на рис. 7.2, *6,* в некоторый момент времени нет ни одного процесса размером от 12 до 16 Мбайт. В результате раздел размером 16 Мбайт будет пустовать, в то время как он мог бы с успехом использоваться меньшими процессами.

Таким образом, более предпочтительным подходом является использование одной очереди для всех процессов (см. рис. 7.3, б*).* В момент, когда требуется загрузить процесс в основную память, для этого выбирается наименьший доступный раздел, способный вместить данный процесс.

Если все разделы заняты, следует принять решение об освобождении одного из них. По-видимому, следует отдать предпочтение процессу, занимающему наименьший раздел, способный вместить загружаемый процесс.

Можно учесть и другие факторы, такие как приоритет процесса или его состояние (заблокирован он или активен).

Использование разделов разного размера по сравнению с использованием разделов одинакового размера придает дополнительную гибкость данному методу.

Кроме того, схемы с фиксированными разделами относительно просты, предъявляют минимальные требования к операционной системе; накладные расходы работы процессора невелики.

Однако у этих схем имеются серьезные недостатки.

- Количество разделов, определенное в момент генерации системы, ограничивает количество активных (не приостановленных) процессов.

- Поскольку размеры разделов устанавливаются заранее, в момент генерации системы, небольшие процессы приводят к неэффективному использованию памяти. В средах, в которых заранее известны потребности в памяти всех задач, применение описанной схемы может быть оправдано, но в большинстве случаев эффективность этой технологии крайне низка.

Фиксированное распределение в настоящее время практически не используется. Примером успешной операционной системы с использованием данной технологии может служить ранняя операционная система IBM для мейнфреймов OS/MFT (мультипрограммная с фиксированным количеством задач - Multiprogrammning with a Fixed number of Tasks).

Динамическое распределение

Для преодоления сложностей, связанных с фиксированным распределением, был разработан альтернативный подход, известный как ***динамическое распределение.***

Этот подход в настоящее время также вытеснен более сложными и эффективными технологиями управления памятью.

При динамическом распределении образуется переменное количество разделов переменной длины. При размещении процесса в основной памяти для него выделяется строго необходимое количество памяти, и не более того.

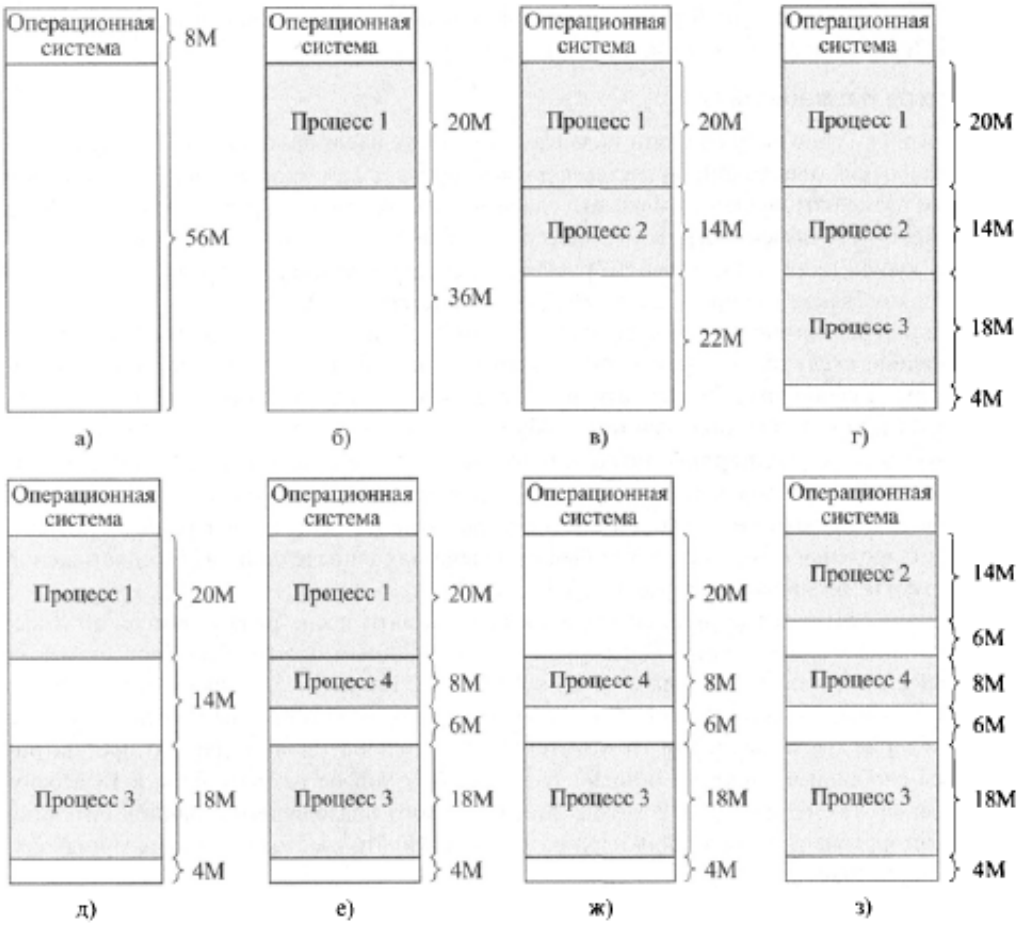


Рис. 7.4. Пример динамического распределения

Этот метод хорошо начинает работу, но плохо продолжает - в конечном счете приводит к наличию множества мелких дыр в памяти. Со временем память становится все более и более фрагментированной и снижается эффективность ее использования. Это явление называется **внешней фрагментацией** (external fragmentation), что отражает тот факт, что сильно фрагментированной становится память, внешняя по отношению ко

всем разделам (в отличие от рассмотренной ранее внутренней фрагментации).

Один из методов преодоления этого явления состоит в **уплотнении** (compaction): время от времени операционная система перемещает процессы в памяти так, чтобы они занимали смежные области памяти; свободная память при этом собирается в один блок.

Например, на рис. 7.4,з после уплотнения памяти мы получим блок свободной памяти размером 16 Мбайт, чего может оказаться вполне достаточно для загрузки нового процесса.

Сложность применения уплотнения состоит в том, что при этом расходуется дополнительное время; кроме того, уплотнение требует динамического перемещения процессов в памяти, т.е. должна быть обеспечена возможность перемещения программы из одной области основной памяти в другую без потери корректности ее обращений к памяти.

Алгоритм размещения

Поскольку уплотнение памяти вызывает дополнительные расходы времени процессора, разработчик операционной системы должен принять разумное решение о том, каким образом размещать процессы в памяти (образно говоря, каким образом затыкать дыры).

Когда наступает момент загрузки процесса в основную память и имеется несколько блоков свободной памяти достаточного размера, операционная система должна принять решение о том, какой именно свободный блок использовать.

Можно рассматривать три основных алгоритма:

1) ***наилучший подходящий***
2) ***первый подходящий***
3) ***следующий подходящий.***

Все они, само собой разумеется, ограничены выбором среди свободных блоков размера, достаточно большого для размещения процесса.

Метод ***наилучшего подходящего*** выбирает блок, размер которого наиболее близок к требуемому; метод ***первого подходящего*** проверяет все свободные блоки с начала памяти и выбирает первый достаточный по размеру для размещения процесса.

Метод следующего подходящего работает так же, как и метод первого подходящего, однако начинает проверку с того места, где был выделен блок в последний раз (по достижении конца памяти он продолжает работу с ее начала).

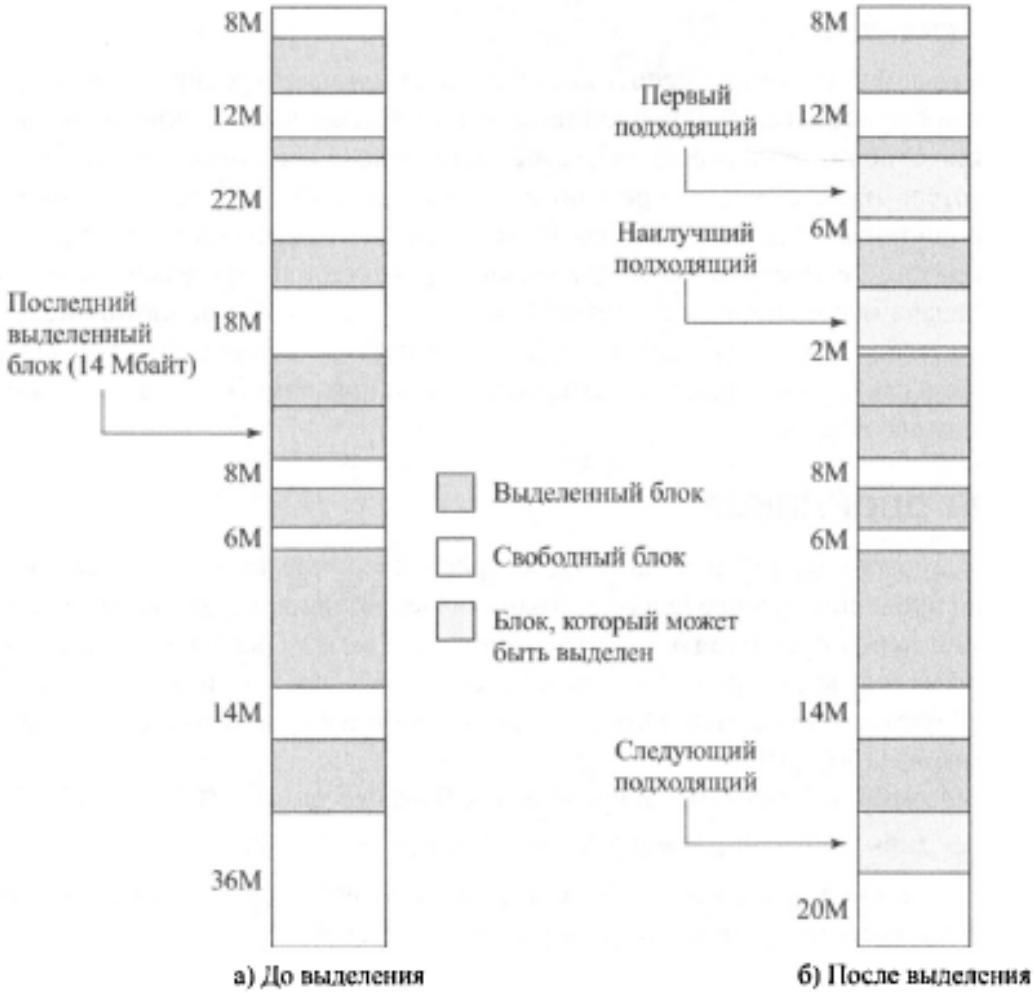


Рис. 7.5. Пример конфигурации памяти до и после выделения блока размером 16 Мбайт

На рис. 7.5.а показан пример конфигурации памяти после ряда размещений и выгрузки процессов из памяти. Последним использованным блоком был блок размером 22 Мбайт, в котором был создан раздел в 14 Мбайт.

На рис. 7.5.б показано различие в технологии наилучшего, первого и следующего подходящего при выполнении запроса на выделение блока размером 16 Мбайт. Метод наилучшего

подходящего просматривает все свободные блоки и выбирает наиболее близкий по размеру блок в 18 Мбайт, оставляя фрагмент размером 2 Мбайт. Метод первого подходящего в данной ситуации оставляет фрагмент свободной памяти размером 6 Мбайт, а метод следующего подходящего - 20 Мбайт.

Какой из этих методов окажется наилучшим, будет зависеть от точной последовательности загрузки и выгрузки процессов и их размеров. Однако можно говорить о некоторых обобщенных выводах.

Обычно *алгоритм первого подходящего* не только проще, но и быстрее и дает лучшие результаты.

Алгоритм *следующего подходящего*, как правило, дает немного худшие результаты. Это связано с тем, что алгоритм следующего подходящего проявляет склонность к более частому выделению памяти из свободных блоков в конце памяти. В результате самые большие блоки свободной памяти (которые обычно располагаются в конце памяти) быстро разбиваются на меньшие фрагменты и, следовательно, при использовании метода следующего подходящего уплотнение должно выполняться чаще.

С другой стороны, алгоритм первого подходящего обычно засоряет начало памяти небольшими свободными блоками, что приводит к увеличению времени поиска подходящего блока в последующем.

Метод ***наилучшего подходящего***, вопреки своему названию, оказывается, как правило, наихудшим. Так как он ищет блоки, наиболее близкие по размеру к требуемому, он оставляет после себя множество очень маленьких блоков. В результате, хотя при каждом выделении впустую тратится наименьшее возможное количество памяти, основная память очень быстро засоряется множеством мелких блоков, неспособных удовлетворить ни один запрос (так что при этом алгоритме уплотнение памяти должно выполняться значительно чаще).

Алгоритм замещения

В многозадачной системе с использованием динамического распределения наступает момент, когда все процессы в основной памяти находятся в заблокированном состоянии, а памяти для дополнительного процесса недостаточно даже после уплотнения.

Чтобы избежать потерь процессорного времени на ожидание деблокирования активного процесса, операционная система может выгрузить один из процессов из основной памяти и, таким образом, освободить место для нового процесса или процесса в состоянии готовности.

Задача операционной системы - определить, какой именно процесс должен быть выгружен из памяти.

Система двойников

Как фиксированное, так и динамическое распределение памяти имеют свои недостатки. Фиксированное распределение ограничивает количество активных процессов и неэффективно использует память при несоответствии между размерами разделов и процессов.

Динамическое распределение реализуется более сложно и включает накладные расходы по уплотнению памяти. Интересным компромиссом в этом плане является система двойников.

В системе двойников память распределяется блоками размером 2^K, $L \le K \le U$, где 2^L - минимальный размер выделяемого блока памяти; 2^U - наибольший распределяемый блок; вообще говоря, 2^U представляет собой размер всей доступной для распределения памяти.

Вначале все доступное для распределения пространство рассматривается как единый блок размером 2^U. При запросе размером *S,* таким, что $2^{U-1} < S \le 2^U$, выделяется весь блок. В противном случае блок разделяется на два эквивалентных двойника с размерами 2^{U-1} . Если $2^{U-2} < S \le 2^{U-1}$, то по запросу выделяется один из двух двойников; в противном случае один из двойников вновь делится пополам.

Этот процесс продолжается до тех пор, пока не будет сгенерирован наименьший блок, размер которого не меньше *S*. Система двойников постоянно ведет список "дыр" (доступных блоков) для 2^i. Дыра может быть удалена из списка (i+1) разделением ее пополам и внесением двух новых дыр размером *i* в список *i*. Когда пара двойников в списке *i* оказывается освобожденной, они удаляются из списка и объединяются в единый блок в списке *(i*+1). Ниже приведен рекурсивный алгоритм для удовлетворения запроса размером $2^{i-1} < S \leq 2^i$, в котором осуществляется поиск дыры размером 2^i .

```
void get_hole(int i)
{
    if (i == (U+1))  < Ошибка >;
    if (< Список i пуст >)
    {
        get_hole(i+1);
        < Разделить дыру на двойники >;
        < Поместить двойники в список i >;
    }
    < Взять первую дыру из списка i >;
}
```

На рис. 7.6 приведен пример использования блока с начальным размером 1 Мбайт.

Первый запрос А - на 100 Кбайт (для него требуется блок размером 128 Кбайт). Для этого начальный блок делится на два двойника по 512 Кбайт. Первый из них делится на двойники размером 256 Кбайт, и, в свою очередь, первый из получившихся при этом разделении двойников также делится пополам. Один из получившихся двойников размером 128 Кбайт выделяется запросу А

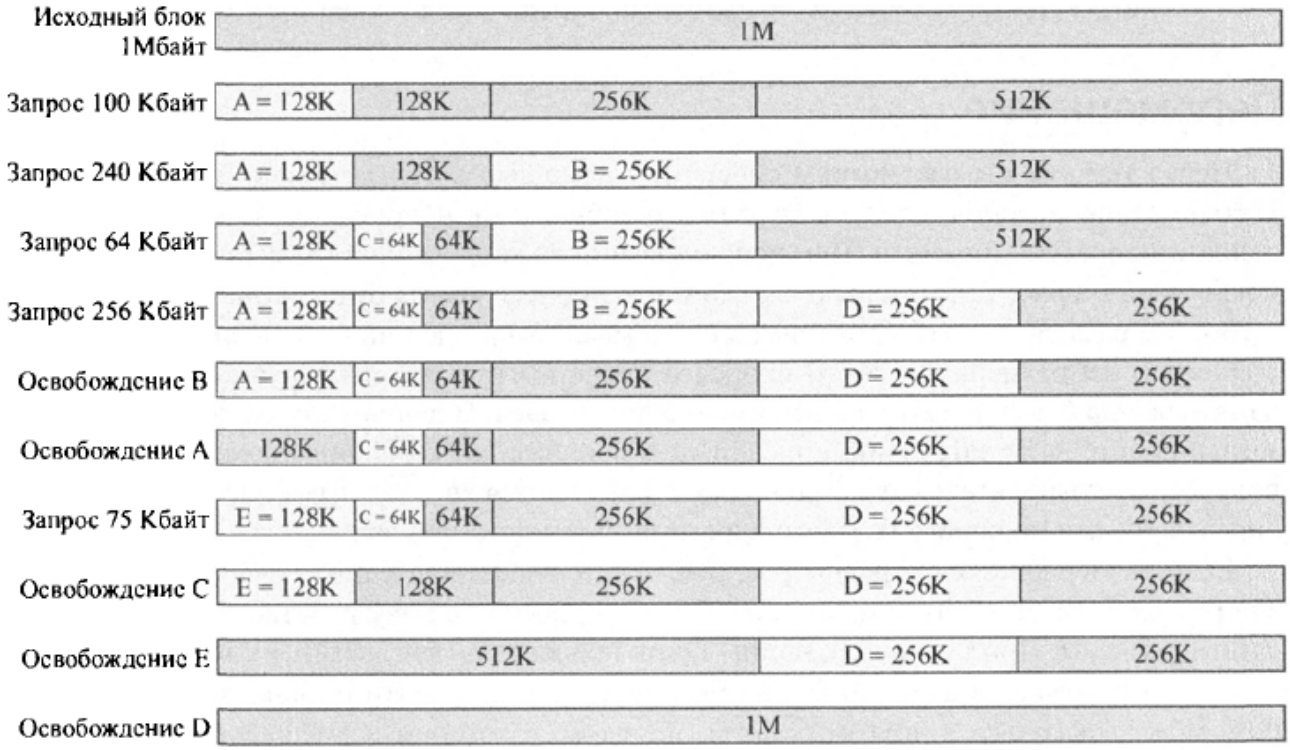


Рис. 7.6. Пример системы двойников

Следующий запрос B требует 256 Кбайт. Такой блок имеется в наличии и выделяется. Процесс продолжается с разделением и слиянием двойников при необходимости. Обратите внимание, что после освобождения блока E происходит слияние двойников по 128 Кбайт в один блок размером 256 Кбайт, который, в свою очередь, тут же сливается со своим двойником.

На рис. 7.7 показано представление системы двойников в виде бинарного дерева, непосредственно после освобождения блока B. Листья представляют текущее распределение памяти. Если два двойника являются листьями , то по крайней мере один из них занят; в противном случае они должны слиться в блок большего размера.

Система двойников представляет собой разумный компромисс для преодоления недостатков схем фиксированного и динамического распределения , но в современных операционных системах ее превосходит виртуальная память, основанная на страничной организации и сегментации. Однако система двойников нашла применение в параллельных системах как эффективное средство распределения и освобождения параллельных программ.

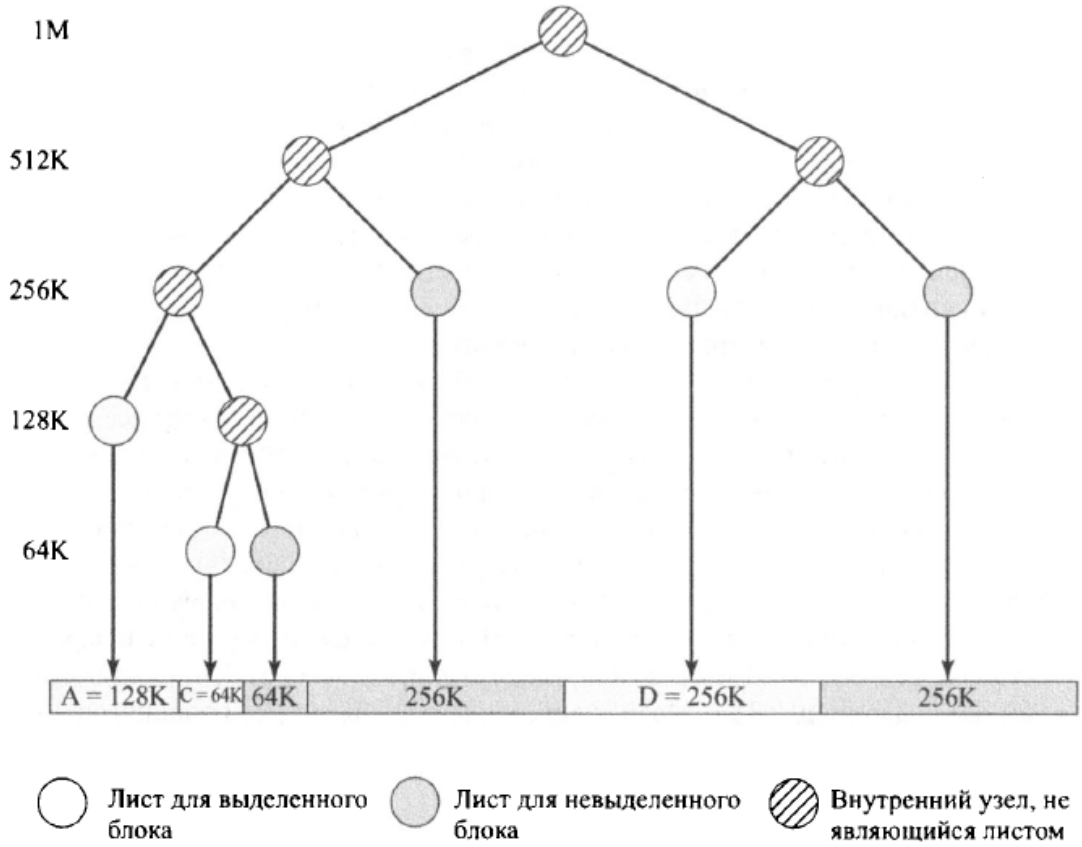


Рис. 7.7. Представление системы двойников в виде бинарного дерева

Модифицированная версия системы двойников используется для распределения памяти ядром UNIX.

Перемещение

При использовании фиксированной схемы распределения, показанной на рис. 7.3, а, можно ожидать, что процесс всегда будет назначаться одному и тому же разделу памяти.

Это означает, что какой бы раздел ни был выбран для нового процесса, для размещения этого процесса после выгрузки и последующей загрузки в память всегда будет использоваться именно этот раздел. В данном случае можно использовать простейший загрузчик: при загрузке процесса все относительные ссылки в коде замещаются абсолютными адресами памяти, определенными на основе базового адреса загруженного процесса.

Если размеры разделов равны и существует единая очередь процессов для разделов разного размера, процесс по ходу работы может занимать разные разделы. При первом создании образа

процесса он загружается в некоторый раздел памяти; позже, после того как он был выгружен из памяти и вновь загружен, процесс может оказаться в другом разделе (не в том, в котором размещался в последний раз). Та же ситуация возможна и при динамическом распределении.

Таким образом, расположение команд и данных, к которым обращается процесс, не является фиксированным и изменяется всякий раз при выгрузке и загрузке (или перемещении) процесса. Для решения этой проблемы следует различать типы адресов.

Логический адрес представляет собой ссылку на ячейку памяти, не зависящую от текущего расположения данных в памяти; перед тем как получить доступ к этой ячейке памяти, необходимо транслировать логический адрес в физический.

Относительный адрес представляет собой частный случай логического адреса, когда адрес определяется положением относительно некоторой известной точки (обычно - начала программы).

Физический адрес (известный также как абсолютный) представляет собой действительное расположение интересующей нас ячейки основной памяти.

Программа, которая использует относительные адреса в памяти, загружается с помощью динамической загрузки времени выполнения.

Обычно все ссылки на память в загруженном процессе даны относительно начала этой программы.

Таким образом, для корректной работы программы требуется аппаратный механизм, который транслировал бы относительные адреса в физические в процессе выполнения команды, которая обращается к памяти.

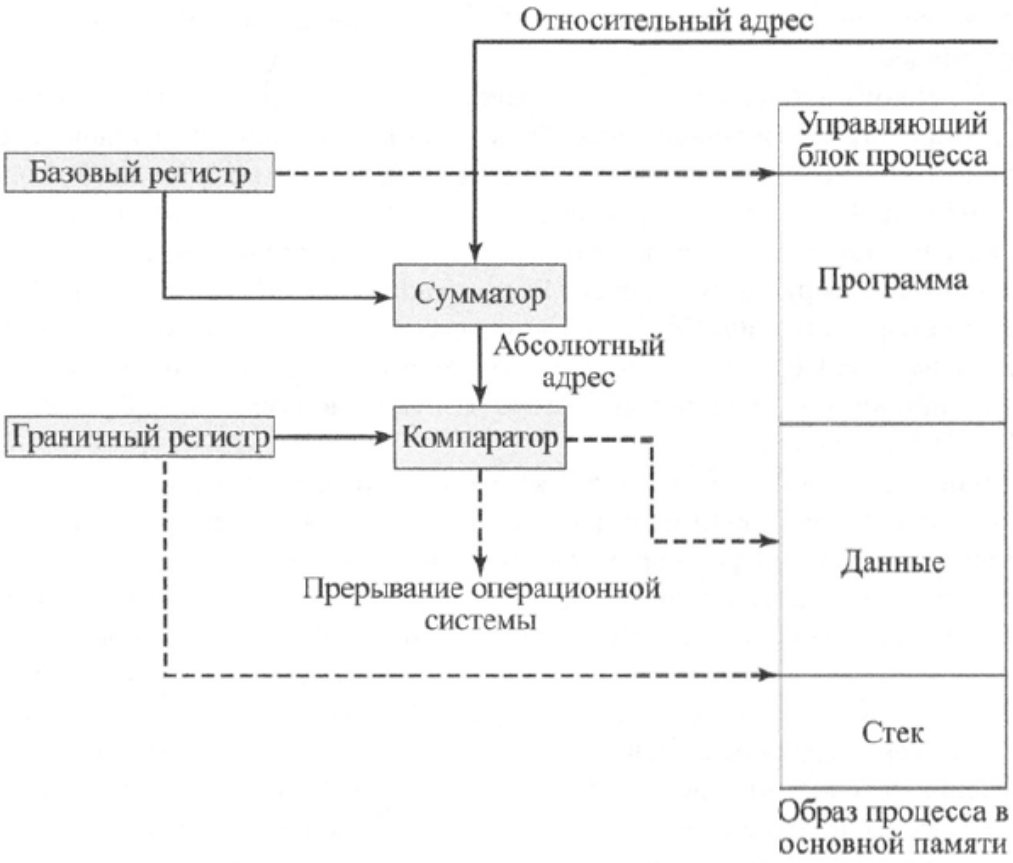


Рис. 7.8. Аппаратная поддержка перемещения

На рис. 7.8 показан обычно используемый способ трансляции адреса. Когда процесс переходит в состояние выполнения, в специальный регистр процессора, иногда называемый базовым, загружается начальный адрес процесса в основной памяти. Кроме того, используется "граничный" (bounds) регистр, в котором содержится адрес последней ячейки памяти программы. Эти значения заносятся в регистры при загрузке программы в основную память.

При выполнении процесса встречающиеся в командах относительные адреса обрабатываются процессором в два этапа.

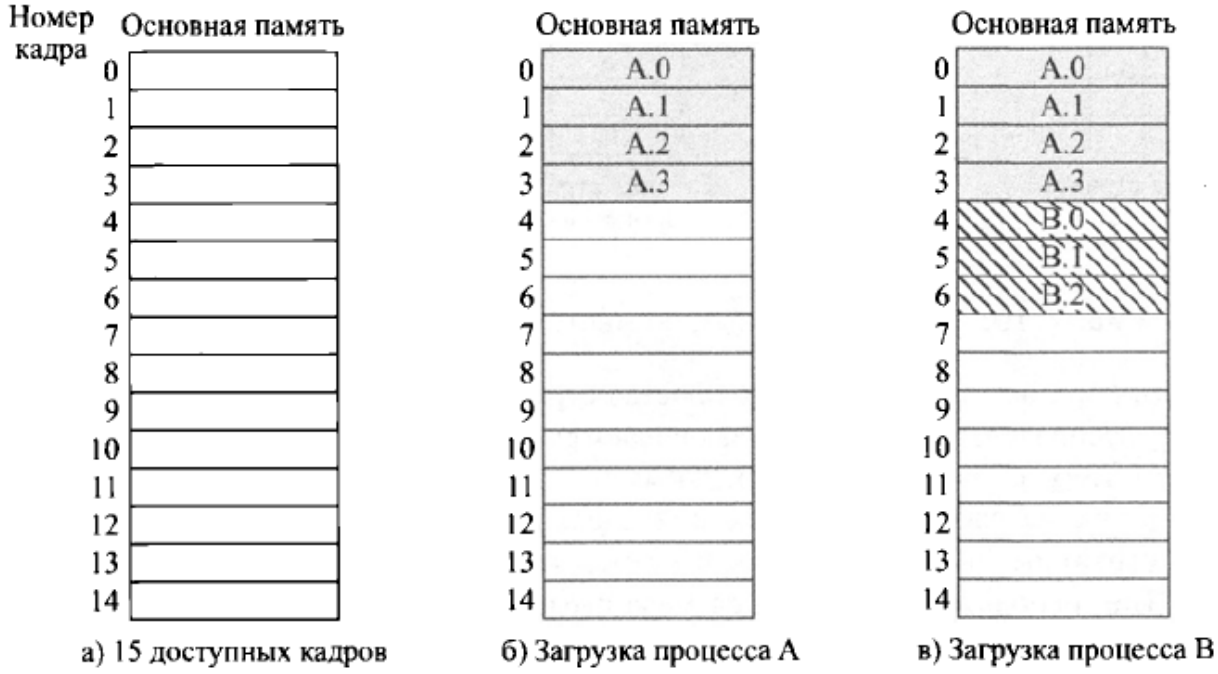
Сначала к относительному адресу прибавляется значение базового регистра для получения абсолютного адреса.

Затем полученный абсолютный адрес сравнивается со значением в граничном регистре. Если полученный абсолютный адрес принадлежит данному процессу, команда может быть выполнена; в противном случае генерируется соответствующее данной ошибке прерывание операционной системы.

Схема, представленная на рис. 7.8, обеспечивает возможность выгрузки и загрузки программ в основную память в процессе их выполнения; кроме того, образ каждого процесса ограничен адресами, содержащимися в базовом и граничном регистрах, и, таким образом, защищен от нежелательного доступа со стороны других процессов.

СТРАНИЧНАЯ ОРГАНИЗАЦИЯ ПАМЯТИ

Как разделы с разными фиксированными размерами, так и разделы переменного размера недостаточно эффективно используют память. Результатом работы первых становится внутренняя фрагментация, результатом работы последних - внешняя. Предположим, однако, что основная память разделена на одинаковые блоки относительно небольшого фиксированного размера. Тогда блоки процесса, известные **как страницы (pages)**, могут быть связаны со свободными блоками памяти, известными как **кадры (frames) или фреймы.** Каждый кадр может содержать одну страницу данных. При такой организации памяти, внешняя фрагментация отсутствует вовсе, а потери из-за внутренней фрагментации ограничены частью последней страницы процесса.



а) 15 доступных кадров б) Загрузка процесса А в) Загрузка процесса В

Рис. 7.9. Распределение страниц по свободным кадрам

На рис. 7.9 показано использование страниц и кадров. В любой момент времени некоторые из кадров памяти используются, а некоторые свободны. Операционная система поддерживает список свободных кадров.

Процесс A, хранящийся на диске, состоит из четырех страниц. Когда приходит время загрузить этот процесс в память, операционная система находит четыре свободных кадра и загружает страницы процесса A в эти кадры (рис. 7.9, 6). Затем загружаются процесс B, состоящий из трех страниц, и процесс C, состоящий из четырех страниц. После этого процесс B приостанавливается и выгружается из основной памяти.

Позже наступает момент, когда все процессы в памяти оказываются заблокированными, и операционная система загружает в память новый процесс D, состоящий из пяти страниц.

Теперь предположим, что, как в только что рассмотренном выше примере, не имеется одной непрерывной области кадров, достаточной для размещения процесса целиком. Помешает ли это операционной системе загрузить процесс D? Нет, поскольку в этой ситуации можно воспользоваться концепцией логических адресов. Однако одного регистра базового адреса в этой ситуации

недостаточно, и для каждого процесса операционная система должна поддерживать таблицу страниц.

Таблица страниц указывает расположение кадров каждой страницы процесса. Внутри программы логический адрес состоит из номера страницы и смещения внутри нее. Вспомним, что в случае простого распределения логический адрес представляет собой расположение слова относительно начала программы, которое процессор транслирует в физический адрес.

При страничной организации преобразование логических адресов в физические также остается задачей аппаратного уровня, решаемой процессором.

Теперь процессор должен иметь информацию о том, где находится таблица страниц текущего процесса. Представленный логический адрес (номер страницы и смещение) процессор превращает с использованием таблицы страниц в физический адрес (номер кадра, смещение).

Рис. 7.10. Структуры данных, соответствующие примеру на рис. 7.9, е

На рис. 7.10 показаны различные таблицы страниц, после того как процесс О оказывается загруженным в страницы 4, 5, 6, 11 и 12. Таблица страниц содержит по одной записи для каждой страницы процесса, так что таблицу легко проиндексировать номером страницы, начиная с О. Каждая запись содержит номер фрейма в основной памяти (если таковой имеется), в котором хранится соответствующая страница. Кроме того, операционная система поддерживает единый список свободных (т. е . не занятых никаким процессом и доступных для размещения в них страниц) кадров.

Таким образом, описанная здесь простая страничная организация подобна фиксированному распределению. Отличия

заключаются в достаточно малом размере разделов, которые к тому же могут не быть смежными. Для удобства работы с такой схемой добавим правило, в соответствии с которым размер страницы (а следовательно, и размер кадра) должен представлять собой степень 2. При использовании такого размера страниц легко показать, что относительный адрес, который определяется относительно начала программы, и логический адрес, представляющий собой номер кадра и смещение, идентичны. Соответствующий пример приведен на рис. 7.11.

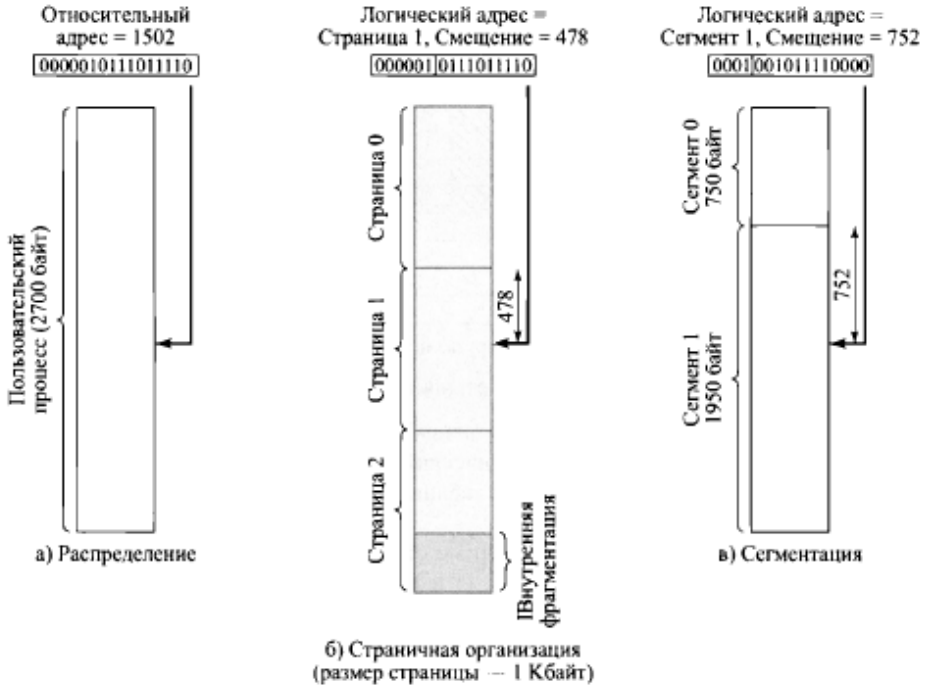


Рис. 7.11. Логические адреса

Здесь используются 16-битный адрес и страницы размером 1 Кбайт = 1024 байт. Относительный адрес 1502 в бинарном виде записывается как 0000010111011110. При размере страницы в 1 Кбайт поле смещения требует 10 бит, оставляя 6 бит для номера страницы.

Таким образом, программа может состоять максимум из $2^6 = 64$ страниц по 1 Кбайт каждая. Как показано на рис. 7.11, б, относительный адрес 1502 соответствует смещению 478 (0111011110) на странице 1 (000001), что дает то же 16-битное число 0000010111011110.

Использование страниц с размером, равным степени двойки, приводит к таким следствиям.

Во-первых, схема логической адресации прозрачна для программиста, ассемблера и компоновщика. Каждый логический адрес (номер страницы и смещение) программы идентичен относительному адресу.

Во-вторых, при этом относительно просто реализуется аппаратная функция преобразования адресов во время работы.

СЕГМЕНТАЦИЯ

Альтернативным способом распределения пользовательской программы является сегментация. В этом случае программа и связанные с ней данные разделяются на ряд сегментов.

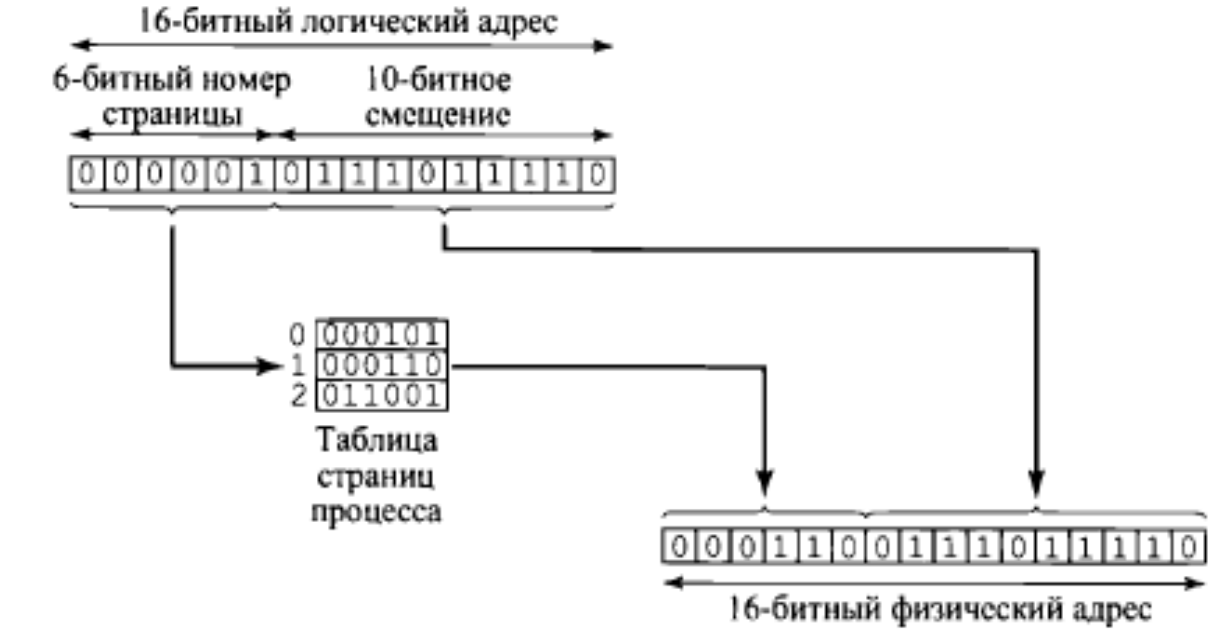
Хотя и существует максимальный размер сегмента, на сегменты не накладывается условие равенства размеров. Как и при страничной организации, логический адрес состоит из двух частей, в данном случае - номера сегмента и смещения. Использование сегментов разного размера делает этот способ похожим на динамическое распределение памяти.

Если не используются оверлеи и виртуальная память, то для выполнения программы все ее сегменты должны быть загружены в память; однако в отличие от динамического распределения в этом случае сегменты могут занимать несколько разделов, которые к тому же могут не быть смежными.

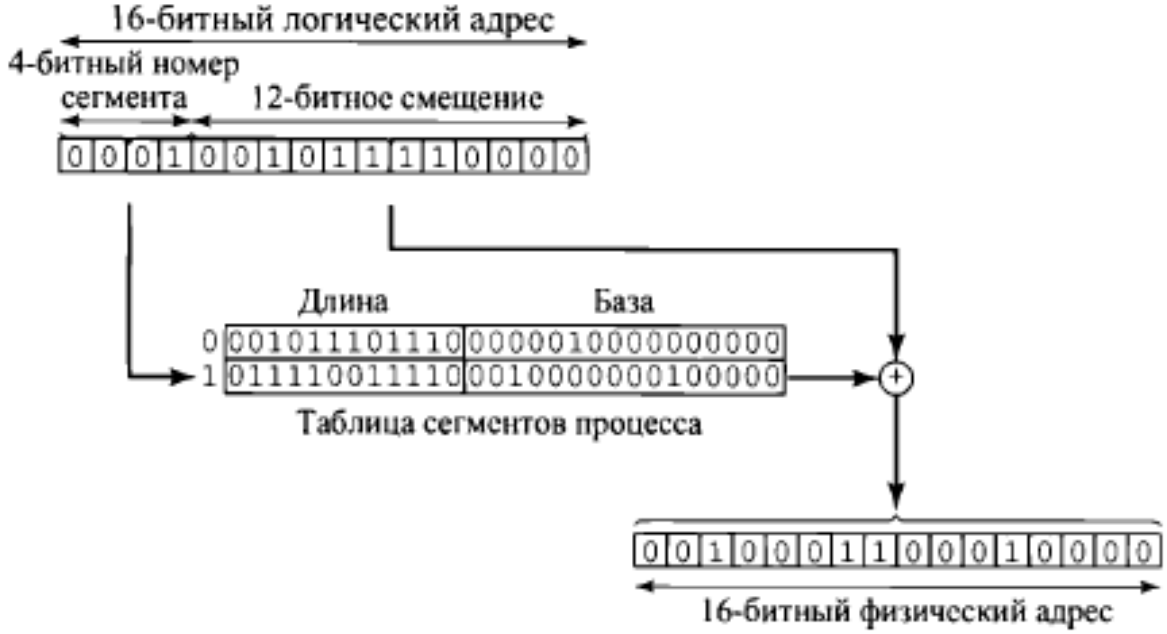
При сегментации устраняется внутренняя фрагментация, однако, как и при динамическом распределении, наблюдается внешняя фрагментация. Тем не менее ее степень снижается, в силу того что процесс разбивается на ряд небольших частей.

В то время как страничная организация невидима для программиста, сегментация, как правило, видима и обычно используется при размещении кода и данных в разных сегментах. При использовании принципов модульного программирования как код, так и данные могут быть дополнительно разбиты на сегменты.

Главным недостатком при работе с сегментами является необходимость заботиться о том, чтобы размер сегмента не превысил максимальный.



а) Страничная организация



б) Сегментация

Рис. 7.12. Примеры трансляции логических адресов в физические