Распределение ресурсов. Управление памятью. Общие вопросы

Лекция

Ревизия: 0.1

Последняя модификация: Vlad Kovtun

© NRJETIX 2000 - 2010

Дата последней модификации: 10.03.2010 23:39:00

История изменений

08.03.2010 – Версия 0.1.   Первичный документ. Ковтун В.Ю.

Содержание

История изменений 2

Содержание 3

Лекция 5. Распределение ресурсов. Управление памятью. Общие вопросы. Часть 1 4

- Вопросы 4
- Введение 4
  - Основы управления памятью 4
  - Моделирование многозадачности 5
  - Настройка адресов и защита 7
- Подкачка 8
  - Управление памятью с помощью битовых массивов 10
  - Управление памятью с помощью связных списков 11
- Виртуальная память 13
  - Страничная организация памяти 13

Литература 16

Лекция 5. Распределение ресурсов. Управление памятью. Общие вопросы. Часть 1

Вопросы

1. Введение.
2. Подкачка.
3. Виртуальная память. Основы.

Введение

Память в компьютерах имеет иерархическую структуру. Небольшая часть ее представляет собой очень быструю, дорогую, энергозависимую кэш-память (несколько уровней). Кроме того, компьютеры обладают десятками мегабайт среднескоростной, имеющей среднюю цену, также энергозависимой оперативной памяти ОЗУ (RAM) и десятками или сотнями гигабайт медленного, дешевого, энергонезависимого пространства на жестком диске. Одной из задач операционной системы является **координация использования всех этих составляющих памяти**.

Часть ОС, отвечающая за управление памятью, называется **модулем управления памятью** или **менеджером памяти**. Он следит за тем, какая часть памяти используется в данный момент, а какая — свободна; при необходимости выделяет память процессам и по их завершении освобождает ресурсы; управляет обменом данных между оперативной памятью и диском, если память слишком мала для того, чтобы вместить все процессы.

Основы управления памятью

Системы управления памятью можно разделить на два класса:

- Перемещающие процессы между оперативной памятью и диском во время их выполнения: осуществляющие подкачку процессов целиком (swapping) или использующие страничную подкачку (paging).
- И те, которые этого не делают.

Коротко остановимся на простейших системах управления памятью.

Однозадачная система без подкачки с диска

Когда ОС организована таким образом, в каждый конкретный момент времени может работать только один процесс. Как только пользователь набирает команду, ОС копирует запрашиваемую программу с диска в память и выполняет ее, а после окончания процесса выводит на экран символ приглашения и ждет новой команды. Получив команду, она загружает новую программу в память, записывая ее поверх предыдущей. Такой подход использовался в MS-DOS v3.1-v6.22.

Рис. 1. Три простейшие модели организации памяти при наличии ОС и одного пользовательского процесса. Существуют также и другие возможные варианты

Многозадачность с фиксированными разделами

Самый легкий способ достижения многозадачности представляет собой простое разделение памяти на n (не обязательно равных) разделов. Такое разбиение выполняется на этапе инициализации/запуске системы, и называется OS/MFT – Multiprogramming with a Fixed number of Tasks для OS/360.

Когда задание поступает в память, его можно расположить во входной очереди к наименьшему разделу, достаточно большому для того, чтобы вместить это задание. Так как в данной схеме размер разделов неизменен, все пространство в разделе, не используемое работающим процессом, пропадает. На рис. 2(а) показано, как выглядит система с фиксированными разделами и отдельными очередями входных заданий.

Недостаток сортировки входящих работ по отдельным очередям становится очевидным, когда к большому разделу нет очереди, в то время как к маленькому выстроилось довольно много задач; в примере на рис. 2, а это разделы 1 и 3.

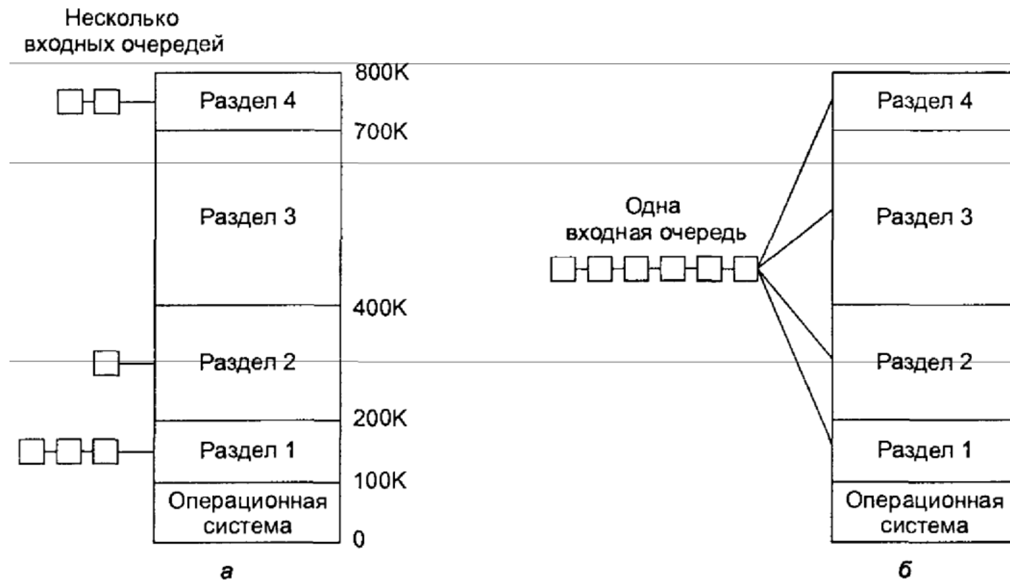


Рис. 2. Фиксированные разделы памяти с отдельными входными очередями для каждого раздела (а); фиксированные разделы памяти с одной очередью на вход (б)

Небольшие задания должны ждать своей очереди, чтобы попасть в память, и это все несмотря на то, что свободна основная часть памяти. **Альтернативная схема** заключается в организации одной общей очереди для всех разделов, как показано на рис. 2(б): как только раздел освобождается, задачу, находящуюся ближе всего к началу очереди и подходящую для выполнения в этом разделе, можно загрузить в него и начать ее обработку. Поскольку нежелательно тратить большие разделы на маленькие задачи, существует другая стратегия: каждый раз после освобождения раздела происходит поиск в очереди наибольшего из помещающихся в этом разделе заданий, и именно это задание выбирается для обработки. Заметим, что последний алгоритм дискриминирует мелкие задачи, как недостойные того, чтобы под них отводился целый раздел, хотя обычно крайне желательно предоставить для наименьших задач (часто интерактивных) лучшее, а не худшее обслуживание.
Одним из выходов из положения можно, создав хотя бы один маленький раздел памяти, который позволит выполнять мелкие задания без долгого ожидания освобождения больших разделов.
При **другом подходе** устанавливается следующее правило: задачу, имеющую право быть выбранной для обработки, можно пропустить не больше k раз. Каждый раз, когда через нее перескакивают, к счетчику добавляется единица. Когда значение счетчика становится равным k, игнорировать задачу более нельзя.

Моделирование многозадачности

При использовании многозадачности повышается эффективность загрузки центрального процессора. Грубо говоря, если средний процесс выполняет вычисления только 20% от того времени, которое он находится в памяти, то при присутствии в памяти одновременно пяти процессов центральный процессор должен быть занят все время. Эта схема слишком оптимистична в отличие от реальной ситуации, поскольку

она предполагает, что все пять процессов никогда не ожидают завершения операции ввода-вывода одновременно.

Более совершенная модель рассматривает эксплуатацию центрального процессора с точки зрения теории вероятности. Предположим, что процесс проводит часть p своего времени в ожидании завершения операции ввода-вывода. Если в памяти находится одновременно n процессов, вероятность того, что все n процессов ждут ввод-вывод (в этом случае центральный процессор будет бездействовать), равна p^n. Тогда степень загрузки центрального процессора будет выражаться формулой:

Степень загрузки центрального процессора $=1-p^n$.

На рис. 3 показана зависимость степени использования центрального процессора от числа n, называемого **степенью многозадачности**.

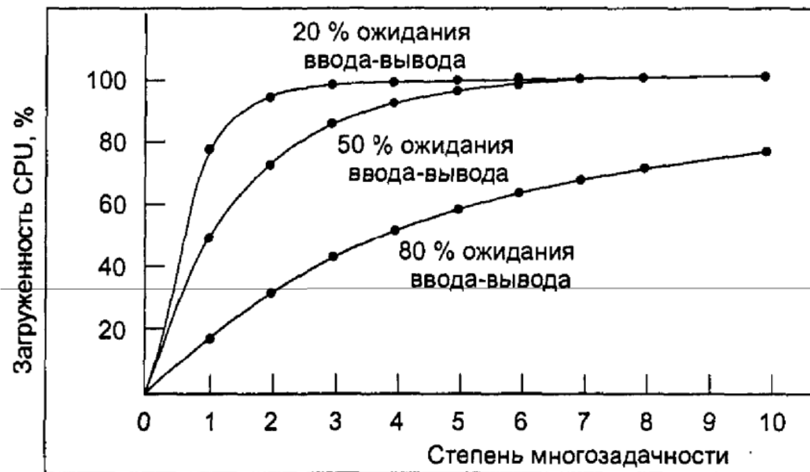


Рис. 3. Зависимость степени загрузки центрального процессора от количества процессов в памяти

Из рис. 3 понятно, что если процессы проводят 80% своего времени в ожидании завершения операции ввода-вывода, то для того, чтобы получить потерю времени процессора ниже 10%, в памяти должны одновременно находиться, по меньшей мере, 10 процессов. Можно представить, что интерактивный процесс, ожидая, пока пользователь напечатает что-либо на терминале, находится в состоянии ожидания ввода-вывода, должно быть ясно, что время ожидания ввода-вывода, равное 80% и больше, не является необычным. Но даже в системах пакетной обработки процессы, выполняющие ввод-вывод в основном с диска, часто имеют такой же или больший процент.

Нужно отметить, что описанная выше вероятностная модель является довольно грубым приближением. **Она неявно предполагает**, что все n процессов независимы, то есть допустима следующая ситуация: в памяти находятся пять процессов, из них три работают, а два ждут. Но когда в системе присутствует один-единственный центральный процессор, он не может одновременно обрабатывать три процесса, поэтому уже готовый к работе процесс обязан ждать освобождения процессора. Таким образом, в реальности процессы не являются независимыми. Более аккуратную модель можно построить с использованием теории организации очередей, но общая идея, на которую мы обратили внимание — многозадачность позволяет процессам использовать центральный процессор тогда, когда при других обстоятельствах он бы бездействовал, — конечно, останется в силе, даже если кривые на рис. 3 немного изменятся.

Хотя модель на рис. 3 очень проста, тем не менее, она позволяет сделать определенный, хотя и приблизительный, прогноз относительно производительности центрального процессора. Например, предположим, что компьютер имеет 32 Мбайт памяти, 16 Мбайт отдано ОС, а каждая программа пользователя занимает по 4 Мбайт. При таких заданных размерах одновременно можно загрузить в память четыре пользовательские программы. При 80% времени на ожидание ввода-вывода в среднем получим загруженность процессора (игнорируя издержки ОС) равной $1-0{,}8^4$ или около 60%. Добавление еще 16 Мбайт памяти позволит ОС повысить степень многозадачности

от четырех до восьми и таким образом повысить степень загрузки процессора до 83%. Другими словами, дополнительные 16 Мбайт увеличат производительность на 38%.

Еще 16 Мбайт могли бы повысить загрузку процессора с 83 до 95 %, таким образом, увеличив производительность всего лишь на 12 %. С помощью этой модели владелец компьютера может решить, что первые 16 Мбайт оперативной памяти — это хорошее вложение капитала, а вторые — нет.

Настройка адресов и защита

Многозадачность вносит две существенные проблемы, требующие решения:

- это настройка адресов для перемещения программы в памяти
- и защита.

Посмотрите на рис. 2: разные задачи будут запущены по различным адресам. Когда программа компонуется (то есть в едином адресном пространстве объединяются основной модуль, написанные пользователем процедуры и библиотечные процедуры), компоновщик должен знать, с какого адреса будет начинаться программа в памяти.

Например, предположим, что первая команда представляет собой вызов процедуры с абсолютным адресом 100 внутри двоичного файла, создаваемого компоновщиком. Если эта программа загружается в раздел 1 (по адресу 100 К), команда обратится к абсолютному адресу 100, который находится внутри ОС. А нужно вызвать процедуру по адресу 100 К + 100. Если же программа загружается во второй раздел, команду нужно переадресовать на 200 К + 100 и т. д. Эта проблема известна как **проблема перемещения программ в памяти или настройки адресов**.

Одним из возможных решений является модификация команд во время загрузки программы в память. В программе, загружаемой в первый раздел, к каждому адресу прибавляется 100 К, в программе, которая загружается во второй раздел, к адресам добавляется 200 К и т. д. Чтобы выполнить подобную настройку адресов во время загрузки, компоновщик должен включить в двоичную программу список или битовый массив с информацией о том, какие слова в программе являются адресами (и их нужно перераспределить), а какие — кодами машинных команд, постоянными или другими частями программы, которые не нужно изменять. Таким образом функционирует ОС OS/MFT.

Настройка адресов во время загрузки не решает проблемы защиты. Вредоносные программы всегда могут создать новую команду и перескочить на нее. Поскольку при такой системе, программы предпочитают использовать абсолютную адресацию памяти, а не адреса относительно какого-либо регистра, не существует способа, который позволил бы запретить программе построение команды, обращающейся к любому слову в памяти для его чтения или записи. В многопользовательских системах крайне нежелательно разрешать процессам чтение или запись в область памяти, принадлежащую другим пользователям.

Для защиты компьютера 360 компания IBM приняла следующее решение: она разделила память на блоки по 2 Кбайт и назначила каждому блоку 4-битовый защитный код. Регистр PSW (Program Status Word — слово состояния программы) содержал 4-битовый ключ. Аппаратура IBM 360 перехватывала все попытки работающих процессов обратиться к любой части памяти, чей защитный код отличался от содержимого регистра слова состояния программы. Так как только ОС могла изменять коды защиты и ключи, предотвращалось вмешательство пользовательских процессов в дела друг друга и в работу ОС.

Альтернативное решение сразу обеих проблем (защиты и перераспределения) заключается в оснащении машины двумя специальными аппаратными регистрами, называемыми **базовым** и **предельным регистрами**. При планировании процесса в базовый регистр загружается адрес начала раздела памяти, а в предельный регистр помещается длина раздела. К каждому автоматически формируемому адресу перед его передачей в память прибавляется содержимое базового регистра. Таким образом, если базовый регистр содержит величину 100К, команда CALL 100 будет превращена в команду CALL 100K+100 без изменения самой команды. Кроме того, адреса проверяются по отношению к предельному регистру для гарантии, что они не используются для адресации памяти вне текущего раздела. Базовый и предельный регистры защищаются аппаратно, чтобы не допустить их изменений пользовательскими программами.

Неудобство этой схемы заключается в том, что требуется выполнять операции сложения и сравнения при каждом обращении к памяти. Операция сравнения может быть выполнена быстро, но сложение — это медленная операция, что обусловлено временем распространения сигнала переноса, за исключением тех случаев, когда употребляется специальная микросхема сложения.

Такая схема использовалась на первом суперкомпьютере в мире CDC 6600. В центральном процессоре Intel 8088 для первых IBM PC применялась упрощенная версия этой модели: были базовые регистры, но отсутствовали предельные. Сейчас такую схему можно встретить лишь в немногих компьютерах.

Подкачка

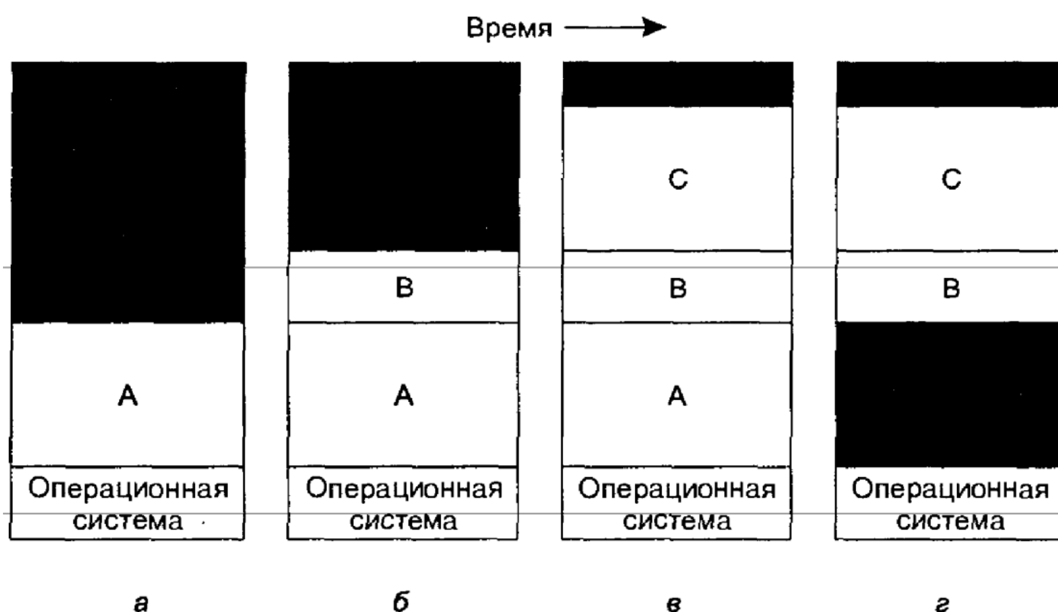
Организация памяти в виде фиксированных разделов проста и эффективна для работы с пакетными системами. Каждое задание после того, как доходит до начала очереди, загружается в раздел памяти и остается там до своего завершения. До тех пор пока в памяти может храниться достаточное количество задач для обеспечения постоянной занятости центрального процессора, нет причин что-либо усложнять.

Но совершенно другая ситуация сложилась с системами разделения времени или персональными компьютерами, ориентированными на работу с графикой. Оперативной памяти иногда оказывается недостаточно для того, чтобы вместить все текущие активные процессы, и тогда избыток процессов приходится хранить на диске, а для обработки динамически переносить их в память.

Существуют два основных подхода к управлению памятью, зависящие (отчасти) от доступного аппаратного обеспечения:

- Самая простая стратегия, называемая **свопингом (swapping) или обычной подкачкой**, заключается в том, что каждый процесс полностью переносится в память, работает некоторое время и затем целиком возвращается на диск.
- Другая стратегия, носящая название **виртуальной памяти**, позволяет программам работать даже тогда, когда они только частично находятся в оперативной памяти.

Работа системы свопинга проиллюстрирована на рис. 5. На начальной стадии в памяти находится только процесс A. Затем создаются или загружаются с диска процессы B и C. На рис. 5(г) процесс A выгружается на диск. Затем появляется процесс D, а процесс B завершается. Наконец, процесс A снова возвращается



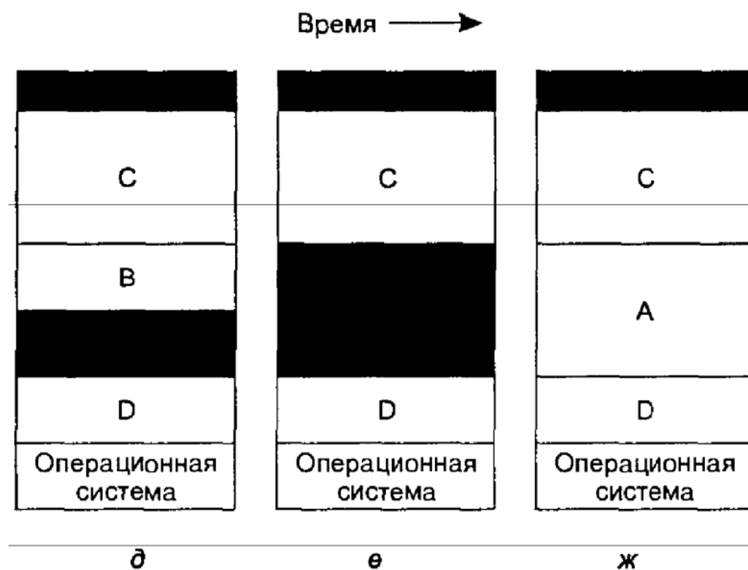


Рис. 4. Распределение памяти изменяется по мере того, как процессы поступают в память и покидают ее. Заштрихованы неиспользуемые области памяти

Основная разница между фиксированными разделами на рис. 2 и непостоянными разделами на рис. 4 заключается в том, что во втором случае количество, размещение и размер разделов изменяются динамически по мере поступления и завершения процессов, тогда как в первом варианте они фиксированы. Гибкость схемы, в которой нет ограничений, связанных с определенным количеством разделов, и каждый из разделов может быть очень большим или совсем маленьким, улучшает использование памяти, но, кроме того, усложняет операции размещения процессов и освобождения памяти, а также отслеживание происходящих изменений.

Когда в результате подкачки процессов с диска в памяти появляется множество неиспользованных фрагментов, их можно объединить в один большой участок, передвинув все процессы в сторону младших адресов настолько, насколько это возможно. Такая операция называется уплотнением или сжатием памяти. Обычно ее не выполняют, потому что на нее уходит много времени работы процессора. Например, на машине с 256 Мбайт оперативной памяти, которая может копировать 4 байта за 40 нс, уплотнение всей памяти займет около 2,7 с.

Еще один момент, на который стоит обратить внимание: сколько памяти должно быть предоставлено процессу, когда он создается или скачивается с диска? Если процесс имеет фиксированный никогда не изменяющийся размер, размещение происходит просто: ОС предоставляет точно необходимое количество памяти, ни больше, ни меньше, чем нужно.

Однако если область данных процесса может расти, например, в результате динамического распределения памяти из кучи, как происходит во многих языках программирования, проблема предоставления памяти возникает каждый раз, когда процесс пытается увеличиться. Когда участок неиспользованной памяти расположен рядом с процессом, его можно отдать в пользу процесса, таким образом, позволив процессу вырасти на размер этого участка. Если же процесс соседствует с другим процессом, для его увеличения нужно или переместить достаточно большой свободный участок памяти, или перекачать на диск один или больше процессов, чтобы создать незанятый фрагмент достаточного размера. Если процесс не может расти в памяти, а область на диске, предоставленная для подкачки, переполнена, процесс будет вынужден ждать освобождения памяти или же будет уничтожен.

Если предположить, что большинство процессов будут увеличиваться во время работы, вероятно, сразу стоит предоставлять им немного больше памяти, чем требуется, а всякий раз, когда процесс скачивается на диск или перемещается в памяти, обрабатывать служебные данные, связанные с перемещением или подкачкой процессов, больше не умещающихся в предоставленной им памяти. Но когда процесс выгружается на диск, должна скачиваться только действительно используемая часть памяти, так как очень расточительно также перемещать и дополнительную память. На рис. 5(а) можно увидеть конфигурацию памяти с предоставлением пространства для роста двух процессов.

Если процесс может иметь два увеличивающихся сегмента, например сегмент данных, используемый как куча для динамически назначаемых и освобождаемых переменных, и сегмент стека для обычных локальных переменных и возвращаемых адресов, предлагается альтернативная схема распределения памяти, показанная на рис. 5(б). Здесь мы видим, что у каждого процесса вверху предоставленной ему области памяти находится стек, который расширяется вниз, и сегмент данных, расположенный отдельно от текста программы, который увеличивается вверх. Область памяти между ними разрешено использовать для любого сегмента. Если ее становится недостаточно, то процесс нужно или перенести на другое, большее свободное место, или выгрузить на диск до появления свободного пространства необходимого размера, или уничтожить.

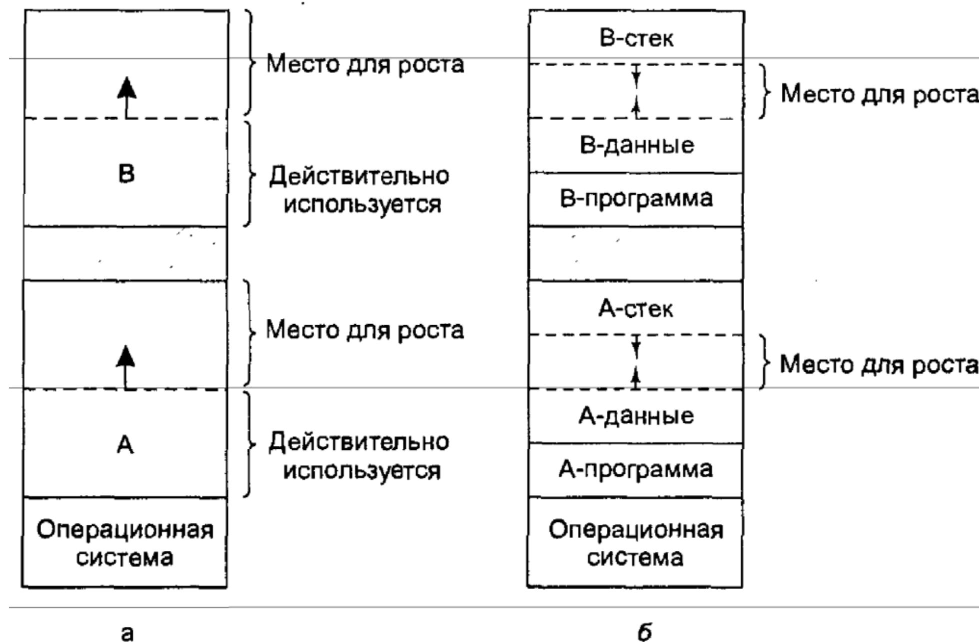


Рис. 5. Предоставление пространства для роста области данных (а); предоставление пространства для роста стека и области данных (б)

Управление памятью с помощью битовых массивов

Если память выделяется динамически, этим процессом должна управлять операционная система. Существует два способа учета использования памяти:

- битовые массивы, иногда называемые битовыми картами,
- и списки свободных участков.

Далее будут рассмотрены оба метода.

При работе с битовым массивом память разделяется на **единичные блоки** размещения размером от нескольких слов до нескольких килобайт. В битовой карте каждому свободному блоку соответствует один бит, равный нулю, а каждому занятому блоку — бит, установленный в 1 (или наоборот). На рис. 6 показана часть памяти и соответствующий ей битовый массив. Черточками отмечены единичные блоки памяти. Заштрихованные области (0 в битовой карте) свободны.

Размер единичного блока представляет собой важный вопрос стадии разработки системы. Чем меньше единичный блок, тем больше потребуется битовый массив. Однако даже при маленьком единичном блоке, равном четырем байтам, для 32 битов памяти потребуется 1 бит в карте. Тогда память размером в $32n$ будет использовать n битов в карте, таким образом, битовая карта займет всего лишь $\frac{1}{33}$ часть памяти. Если выбираются большие единичные блоки, битовая карта становится меньше, но при этом может теряться существенная часть памяти в последнем блоке каждого процесса (если размер процесса не кратен размеру единичного блока).

Битовый массив предоставляет простой способ отслеживания слов в памяти фиксированного объема, потому что размер битовой карты зависит только от размеров памяти и единичного блока. Основная проблема, возникающая при этой схеме, заключается в том, что при решении переместить k-блочный процесс в память модуль управления памяти должен найти в битовой карте серию из k следующих друг за

другом нулевых битов. Поиск серии заданной длины в битовой карте является медленной операцией (так как искомая последовательность битов может пересекать границы слов в битовом массиве). В этом состоит аргумент против битовых карт.

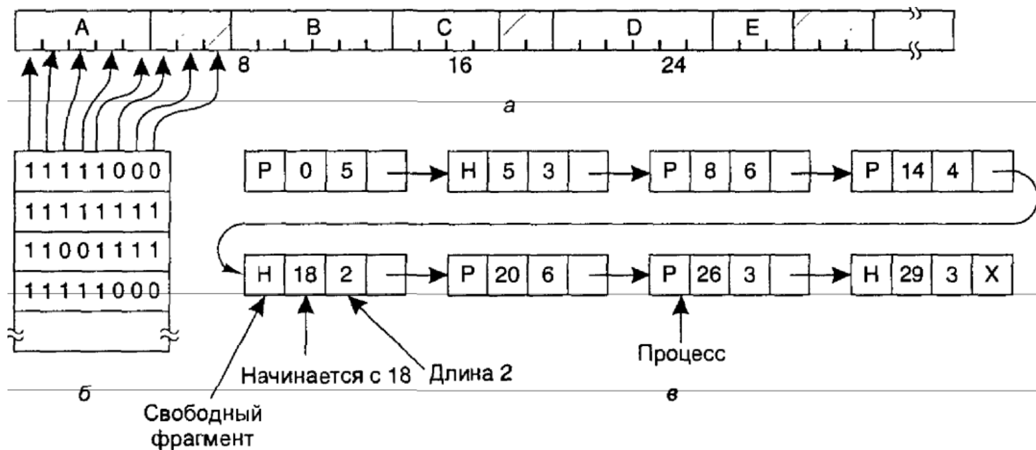


Рис. 6. Часть памяти с пятью процессами и тремя свободными областями (а); соответствующая битовая карта (б); та же информация в виде списка (в)

Управление памятью с помощью связных списков

Другой способ отслеживания состояния памяти предоставляет поддержка связных списков занятых и свободных фрагментов памяти, где сегментом является или процесс, или участок между двумя процессами. Память, показанная на рис. 6, а, представлена в виде связного списка сегментов на рис. 6(в). Каждая запись в списке указывает:

- является ли область памяти свободной (Н, от hole — дыра) или занятой процессом (P - process);
- адрес, с которого начинается эта область;
- длину области памяти;
- содержит указатель на следующую запись.

В примере, список отсортирован по адресам. Такая сортировка имеет следующее **преимущество**: когда процесс завершается или скачивается на диск, изменение списка представляет собой несложную операцию. Закончившийся процесс обычно имеет двух соседей (кроме тех случаев, когда он находится на самом верху или на дне памяти). Соседями могут быть процессы или свободные фрагменты, что приводит к четырем комбинациям, показанным на рис. 7. На рис. 7(а) корректировка списка требует замены Р на Н. На рис. 7(б), в две записи соединяются в одну, а список становится на запись короче. На рис. 7(г) объединяются три записи, а из списка удаляются два пункта. Так как ячейка таблицы процессов для завершившегося процесса обычно будет непосредственно указывать на запись в списке для этого процесса, возможно, удобнее иметь список с двумя связями, чем с одной (последний показан на рис. 7(в)). Такая структура упрощает поиск предыдущей записи и оценку возможности соединения.

Рис. 7. Комбинации соседей для завершения процесса X

Если процессы и свободные участки хранятся в списке, отсортированном по адресам, существует несколько алгоритмов для предоставления памяти процессу, создаваемому

заново (или для существующих процессов, скачиваемых с диска). Допустим, менеджер памяти знает, сколько памяти нужно предоставить. Простейший алгоритм представляет собой **выбор «первого подходящего участка»**. Менеджер памяти просматривает список областей до тех пор, пока не находит достаточно большой свободный участок. Затем этот участок делится на две части: одна отдается процессу, а другая остается неиспользуемой. Так происходит всегда, **кроме статистически нереального случая** точного соответствия свободного участка и процесса. Это быстрый алгоритм, потому что поиск уменьшен настолько, насколько возможно.

Алгоритм «**следующий подходящий участок**» действует с минимальными отличиями от правила «первый подходящий». Он работает так же, как и первый алгоритм, но всякий раз, когда находит соответствующий свободный фрагмент, он запоминает его адрес. И когда алгоритм в следующий раз вызывается для поиска, он стартует с того самого места, где остановился в прошлый раз вместо того, чтобы каждый раз начинать поиск с начала списка, как это делает алгоритм «первый подходящий». **Моделирование работы алгоритма, показало, что производительность схемы «следующий подходящий» немного хуже, чем «первый подходящий»**.

Другой хорошо известный алгоритм называется «**самый подходящий участок**». Он выполняет поиск по всему списку и выбирает наименьший по размеру подходящий свободный фрагмент. Вместо того чтобы делить большую незанятую область, которая может понадобиться позже, этот алгоритм пытается найти участок, близко подходящий к действительно необходимым размерам.

Чтобы привести пример работы алгоритмов «первый подходящий» и «самый подходящий», снова обратимся к рис. 6. Если необходим блок размером 2, правило «первый подходящий» предоставит область по адресу 5, а схема «самый подходящий» разместит процесс в свободном фрагменте по адресу 18.

Алгоритм «самый подходящий» медленнее «первого подходящего», потому что каждый раз он должен производить поиск во всем списке. Но, он выдает еще более плохие результаты, чем «первый подходящий» или «следующий подходящий», поскольку стремится заполнить память очень маленькими, бесполезными свободными областями, то есть фрагментирует память. Алгоритм «первый подходящий» в среднем создает большие свободные участки.

Пытаясь решить проблему разделения памяти на практически точно совпадающие с процессом области и маленькие свободные фрагменты, можно задуматься об алгоритме «**самый неподходящий участок**». Он всегда выбирает самый большой свободный участок, от которого после разделения остается область достаточного размера и ее можно использовать в дальнейшем. Однако моделирование показало, что это также не очень хорошая идея.

Все четыре алгоритма можно ускорить, если поддерживать отдельные списки для процессов и свободных областей. Тогда поиск будет производиться только среди незанятых фрагментов. Неизбежная цена, которую нужно заплатить за увеличение скорости при размещении процесса в памяти, заключается в дополнительной сложности и замедлении при освобождении областей памяти, так как ставший свободным фрагмент необходимо удалить из списка процессов и вставить в список незанятых участков.

Если для процессов и свободных фрагментов поддерживаются отдельные списки, то последний можно отсортировать по размеру, тогда алгоритм «самый подходящий» будет работать быстрее. Когда он выполняет поиск в списке свободных фрагментов от самого маленького к самому большому, то, как только находит подходящую незанятую область, алгоритм уже знает, что она — наименьшая из тех, в которых может поместиться задание, то есть наилучшая. В отличие от схемы с одним списком, дальнейший поиск не требуется. Таким образом, **если список свободных фрагментов отсортирован по размеру, схемы «первый подходящий» и «самый подходящий» одинаково быстры**, а алгоритм «следующий подходящий» не имеет смысла.

При поддержке отдельных списков для процессов и свободных фрагментов возможна небольшая оптимизация. Вместо создания отдельного набора структур данных для списка свободных участков, как это сделано на рис. 6(в), можно использовать сами свободные области. Первое слово каждого незанятого фрагмента может содержать размер фрагмента, а второе слово может указывать на следующую запись. Узлы списка на рис. 7(в), для которых требовались три слова и один бит (Р/Н), больше не нужны.

Еще один алгоритм распределения называется «**быстрый подходящий**», он поддерживает отдельные списки для некоторых из наиболее часто запрашиваемых размеров. Например, могла бы существовать таблица с n записями, в которой первая запись указывает на начало списка свободных фрагментов размером 4 Кбайт, вторая запись является указателем на список незанятых областей размером 8 Кбайт, третья — 12 Кбайт и т. д. Свободный фрагмент размером, скажем, 21 Кбайт, мог бы располагаться или в списке областей 20 Кбайт или в специальном списке участков дополнительных размеров. При использовании правила «быстрый подходящий» поиск фрагмента требуемого размера происходит чрезвычайно быстро. Но этот алгоритм **имеет тот же самый недостаток**, что и все схемы, которые сортируют свободные области по размеру, а именно: если процесс завершается или выгружается на диск, поиск его соседей с целью узнать, возможно ли их соединение, является дорогой операцией. А если не производить слияния областей, память очень скоро окажется разбитой на огромное число маленьких свободных фрагментов, в которые не поместится ни один процесс.

Виртуальная память

Уже достаточно давно люди впервые столкнулись с проблемой размещения программ, оказавшихся слишком большими и поэтому не помещавшихся в доступной физической памяти. Обычно принималось решение о разделении программы на части, называемые **оверлеями (overlays)**. Оверлей 0 обычно запускался первым. После окончания своего выполнения он вызывал следующий оверлей. Некоторые оверлейные системы были очень сложными, позволяющими одновременно находиться в памяти нескольким оверлеям. Оверлеи хранились на диске и по мере необходимости динамически перемещались между памятью и диском средствами ОС.

Несмотря на то, что фактическая работа по загрузке оверлеев с диска и выгрузке на диск выполнялась системой, делить программы на части должен был программист. Разбиение больших программ на оверлеи требовало много времени и было не слишком интересным занятием. Однако такая ситуация продолжалась недолго, так как вскоре был реализован способ выполнения этой работы компьютером.

Разработанный метод известен как **виртуальная память**. **Основная идея виртуальной памяти** заключается в том, что объединенный размер программы, данных и стека может превысить количество доступной физической памяти. ОС хранит части программы, использующиеся в настоящий момент, в ОЗУ, остальные — на диске. Например, программа размером 16 Мбайт сможет работать на машине с 4 Мбайт памяти, если тщательно продумать, какие 4 Мбайт должны храниться в памяти в каждый момент времени. При этом части программы, находящиеся на диске и в памяти, будут меняться местами по мере необходимости.

Виртуальная память может также работать в многозадачной системе при одновременно находящихся в памяти частях многих программ. Когда программа ждет перемещения в память очередной ее части, она находится в состоянии ожидания ввода-вывода и не может работать, поэтому центральный процессор может быть отдан другому процессу тем же самым способом, как в любой другой многозадачной системе.

Страничная организация памяти

Большинство систем виртуальной памяти используют технику, называемую **страничной организацией памяти (paging)**. На любом компьютере существует множество адресов в памяти, к которым может обратиться программа:

```
MOV REG, [1000]
```

она делает это для того, чтобы скопировать содержимое памяти по адресу 1000 в регистр `REG` (или наоборот, в зависимости от архитектуры процессора). Адреса могут формироваться с использованием: индексации, базовых регистров, сегментных регистров и т.д.

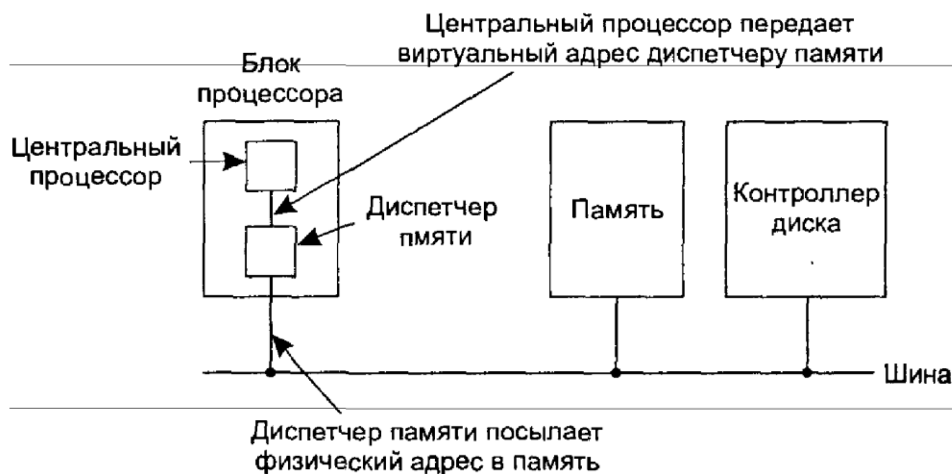


Рис. 8. Расположение и функции диспетчера памяти (MMU). Здесь MMU показан как часть микросхемы процессора

Эти программно формируемые адреса, называемые виртуальными адресами, формируют виртуальное адресное пространство. На компьютерах без виртуальной памяти виртуальные адреса подаются непосредственно на шину памяти и вызывают для чтения или записи слово в физической памяти с тем же самым адресом. Когда используется виртуальная память, виртуальные адреса не передаются напрямую шиной памяти. Вместо этого они передаются диспетчеру памяти (MMU — Memory Management Unit), который отображает виртуальные адреса на физические адреса памяти, как продемонстрировано на рис. 8.

Очень простой пример того, как работает отображение, приведен на рис. 2. Мы рассматриваем компьютер, который может формировать 16-разрядные адреса, от 0 до 64 К. Это виртуальные адреса. Однако у этого компьютера только 32 Кбайт физической памяти, поэтому, хотя программы размером 64 Кбайт могут быть написаны, они не могут целиком быть загружены в память и запущены. Полная копия образа памяти программы размером до 64 Кбайт должна присутствовать на диске, но в таком виде, чтобы ее можно было по мере надобности переносить в память по частям.

Пространство виртуальных адресов разделено на области, называемые **страницами**. Соответствующие области в физической памяти называются **страничными блоками (page frame)**. Страницы и их блоки имеют всегда одинаковый размер. В этом примере они равны 4 Кбайт, но в реальных системах использовались размеры страниц от 512 байт до 64 Кбайт. Имея 64 Кбайт виртуального адресного пространства и 32 Кбайт физической памяти, мы получаем 16 виртуальных страниц и 8 страничных блоков. Передача данных между ОЗУ и диском всегда происходит в страницах.

Когда программа пытается получить доступ к адресу 0, например, используя команду

```
MOV REG, [0]
```

виртуальный адрес 0 передается MMU, который видит, что этот виртуальный адрес попадает на страницу 0 (от 0 до 4095), которая отображается страничным блоком 2 (от 8192 до 12287). MMU виртуальный адрес 0 в физический адрес 8192 и выставляет последний на шину. Память ничего не знает о MMU и видит просто запрос на чтение или запись слова по адресу 8192, который и выполняет. Таким образом, MMU эффективно отображает все виртуальные адреса между 0 и 4095 на физические адреса от 8192 до 12287.

Рис. 9. Связь между виртуальными и физическими адресами, получаемая с помощью таблицы страниц

Точно так же инструкция

`MOV REG, [8192]` преобразуется в команду

`MOV REG, [24576]`

поскольку виртуальный адрес 8192 находится на виртуальной странице 2, а эта страница отображается на физический страничный блок 6 (физические адреса от 24576 до 28671). В качестве третьего примера рассмотрим виртуальный адрес 20500, который адресует 20-й байт от начала виртуальной страницы 5 (виртуальные адреса от 20480 до 24575) и отображается на физический адрес 12288 + 20 = 12308.

Сама по себе возможность отображения 16 виртуальных страниц на любой из восьми страничных блоков с помощью установки соответствующей карты в MMU **не решает проблемы**, заключающейся в том, что размер виртуального адресного пространства больше физической памяти. Так как у нас есть только восемь физических страничных блоков, только восемь виртуальных страниц на рис. 9 воспроизводятся в физической памяти. Другие страницы, обозначенные на рисунке крестиками, не отображаются. В фактическом аппаратном обеспечении страницы, физически присутствующие в памяти, отслеживаются с помощью бита присутствия/отсутствия.

Что происходит, если программа пытается воспользоваться неотображаемой страницей, например, с помощью инструкции

`MOV REG, [32780]`

которая обращается к байту 12 на виртуальной странице 8 (начинающейся с адреса 32768)? MMU замечает, что страница не отображается (обозначена крестиком на рисунке), и инициирует прерывание центрального процессора, передающее управление ОС. Такое прерывание называется **ошибкой из-за отсутствия страницы или страничным прерыванием (page fault)**. ОС выбирает малоиспользуемый страничный блок и записывает его содержимое на диск. Затем она считывает с диска страницу, на которую произошла ссылка, в только что освободившийся блок, изменяет карту отображения и запускает заново прерванную команду.

Например, если ОС решает удалить из ОЗУ страничный блок 1, она загружает виртуальную страницу 8 по физическому адресу 4 K и производит два изменения в карте MMU. Во-первых, отмечается содержимое виртуальной страницы 1 как неотображаемое для того, чтобы перехватывать в будущем любые попытки обращения к виртуальным адресам между 4 K и 8 K. Затем заменяется крест в записи для виртуальной страницы 8 на номер 1, так что когда прерванная команда будет

выполняться заново, она отобразит виртуальный адрес 32780 на физический адрес 4108.

Теперь рассмотрим MMU изнутри, чтобы увидеть, как он работает, и понять, почему мы выбрали размер страницы, являющийся степенью числа 2. На рис. 3 представлен пример виртуального адреса 8196 (0010000000000100 в двоичном виде), который отображается с использованием карты MMU на рис. 3. Входящий 16-разрядный виртуальный адрес разделяется на 4-разрядный номер страницы и 12 битов смещения. При 4 битах под номер страницы в нашей системе может существовать 16 страниц, а с 12 битами смещения мы можем адресоваться ко всем 4096 байтам внутри страницы.

Номер страницы используется в качестве индекса в таблице страниц, выдающей номер страничного блока, соответствующего виртуальной странице. Если бит Присутствия/Отсутствия равен 0, управление переходит к ОС. Если этот бит равен 1, то номер страничного блока, найденный в таблице страниц, записывается в три старших бита выходного регистра, а 12 битов смещения копируются без изменения из входящего виртуального адреса. Все вместе они составляют 15-разрядный физический адрес. Затем выходной регистр помещается на шину памяти как адрес физической памяти.

(Продолжение в следующей лекции 5.2)

Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. – 1040 с.

2. А. Шоу. Логическое проектирование операционных систем. Пер. с англ. –М.: Мир, 1981. –360 с.

3. С. Кейслер. Проектирование операционных систем для малых ЭВМ: Пер. с англ. –М.: Мир, 1986. –680 с.

4. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.

5. Microsoft Development Network. URL: http://msdn.com