**6.1. Организация памяти современного компьютера**

Со времен создания ЭВМ фон Неймана основная память в компьютерной системе организована как линейное (одномерное) адресное пространство, состоящее из последовательности слов, а позже байтов [10]. Аналогично организована и внешняя память. Хотя такая организация и отражает особенности используемого аппаратного обеспечения, она не соответствует способу, которым обычно создаются программы. Большинство программ организованы в виде модулей, некоторые из которых неизменны (только для чтения, только для исполнения), а другие содержат данные, которые могут быть изменены.

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

1. Модули могут быть созданы и скомпилированы независимо друг от друга, при этом все ссылки из одного модуля в другой разрешаются системой во время работы программы.
2. Разные модули могут получать разные степени защиты (только чтение, только исполнение) за счет весьма умеренных накладных расходов.
3. Возможно применение механизма, обеспечивающего совместное использование модулей разными процессами (для случая сотрудничества процессов в работе над одной задачей).

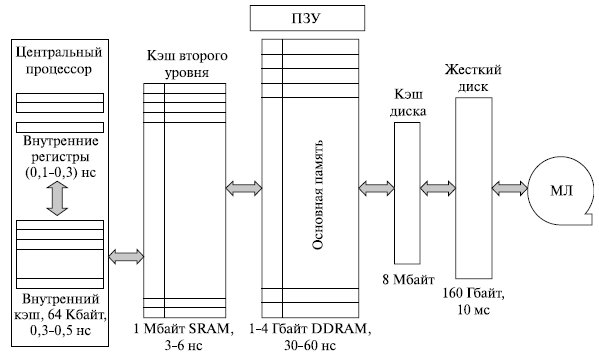
Память – важнейший ресурс вычислительной системы, требующий эффективного управления. Несмотря на то, что в наши дни память среднего домашнего компьютера в тысячи раз превышает память больших ЭВМ 70-х годов, программы увеличиваются в размере быстрее, чем память. Достаточно сказать, что только операционная система занимает сотни Мбайт (например, Windows 2000 – до 30 млн строк), не говоря о прикладных программах и базах данных, которые могут занимать в вычислительных системах десятки и сотни Гбайт.

Перефразированный закон Паркинсона гласит: "Программы расширяются, стремясь заполнить весь объем памяти, доступный для их поддержки" (сказано это было об ОС). В идеале программисты хотели бы иметь неограниченную в размере и скорости память, которая была бы энергонезависимой, т.е. сохраняла свое содержимое при выключении электричества, а также недорого бы стоила. Однако реально пока такой памяти нет. В то же время на любом этапе развития технологии производства запоминающих устройств действуют следующие достаточно устойчивые соотношения:

* чем меньше время доступа, тем дороже бит;
* чем выше емкость, тем ниже стоимость бита;
* чем выше емкость, тем больше время доступа.

Чтобы найти выход из сложившийся ситуации, необходимо опираться не на отдельно взятые компоненты или технологию, а выстроить иерархию запоминающих устройств, показанную на рис. 6.1. При перемещении слева направо происходит следующее :

* снижается стоимость бита;
* возрастает емкость;
* возрастает время доступа;
* снижается частота обращений процессора к памяти.



**Рис. 6.1.**  Иерархия памяти

Предположим, процессор имеет доступ к памяти двух уровней. На первом уровне содержится Е1 слов, и он характеризуется временем доступа Т1 = 1 нс. К этому уровню процессор может обращаться непосредственно. Однако если требуется получить слово, находящееся на втором уровне, то его сначала нужно передать на первый уровень. При этом передается не только требуемое слово, а блок данных, содержащий это слово. Поскольку адреса, к которым обращается процессор, имеют тенденцию собираться в группы (циклы, подпрограммы), процессор обращается к небольшому повторяющемуся набору команд. Таким образом, работа процессора с вновь полученным блоком памяти будет проходить достаточно длительное время.

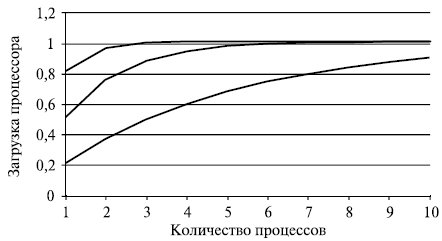
Обозначим через Т2 = 10 нс время обращения ко второму уровню памяти, а через Р – отношение числа нахождений нужного слова в быстрой памяти к числу всех обращений. Пусть в нашем примере Р = 0,95 (т.е. 95% обращений приходится на быструю память, что вполне реально), тогда среднее время доступа к памяти можно записать так:

Tср = 0,95\*1нс + 0,05\* (1нс+10нс)=1,55нс

Этот принцип можно применять не только к памяти с двумя уровнями. Реально так и происходит. Объем оперативной памяти существенно сказывается на характере протекания вычислительного процесса, так как он ограничивает число одновременно выполняющихся программ, т.е. уровень мультипрограммирования. Если предположить, что процесс проводит часть р своего времени в ожидании завершения операции ввода-вывода, то степень загрузки Z центрального процессора (ЦП) в идеальном случае будет выражаться зависимостью

Z = 1 - pn, где n – число процессов.

На рис. 6.2 показана зависимость Z=p(n) для различного времени ожидания завершения операции ввода-вывода (20%, 50% и 80%) и числа процессов n. Большое количество задач, необходимое для высокой загрузки процессора, требует большого объема оперативной памяти. В условиях, когда для обеспечения приемлемого уровня мультипрограммирования имеющейся памяти недостаточно, был предложен метод организации вычислительного процесса, при котором образы некоторых процессов целиком или частично временно выгружаются на диск.



**Рис. 6.2.**  Загрузка процессора при различном числе процессов

Очевидно, что имеет смысл временно выгружать неактивные процессы, находящиеся в ожидании каких-либо ресурсов, в том числе очередного кванта времени центрального процессора. К моменту, когда пройдет очередь выполнения выгруженного процесса, его образ возвращается с диска в оперативную память. Если при этом обнаруживается, что свободного места в оперативной памяти не хватает, то на диск выгружается другой процесс.

Такая подмена (виртуализация) оперативной памяти дисковой памятью позволяет повысить уровень мультипрограммирования, поскольку объем оперативной памяти теперь не столь жестко ограничивает число одновременно выполняемых процессов. При этом суммарный объем оперативной памяти, занимаемой образами процессов, может существенно превосходить имеющийся объем оперативной памяти.

В данном случае в распоряжение прикладного программиста предоставляется виртуальная оперативная память, размер которой намного превосходит реальную память системы и ограничивается только возможностями адресации используемого процесса (в ПК на базе Pentium 232 = 4 Гбайт). Вообще виртуальным (кажущимся) называется ресурс, обладающий свойствами (в данном случае большой объем ОП), которых в действительности у него нет.

Виртуализация оперативной памяти осуществляется совокупностью аппаратных и программных средств вычислительной системы (схемами процессора и операционной системой) автоматически без участия программиста и не сказывается на логике работы приложения.

Виртуализация памяти возможна на основе двух возможных подходов [17]:

* свопинг (swapping) – образы процессов выгружаются на диск и возвращаются в оперативную память целиком;
* виртуальная память (*virtual memory*) – между оперативной памятью и диском перемещаются части образов (сегменты, страницы, блоки и т.п.) процессов.

Недостатки свопинга:

* избыточность перемещаемых данных и отсюда замедление работы системы и неэффективное использование памяти;
* невозможность загрузить процесс, виртуальное пространство которого превышает имеющуюся в наличии свободную память.

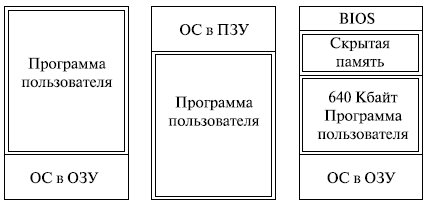
Достоинство свопинга по сравнению с виртуальной памятью – меньшие затраты времени на преобразование адресов в кодах программ, поскольку оно делается один раз при загрузке с диска в память (однако это преимущество может быть незначительным, т.к. выполняется при очередной загрузке только часть кода и полностью преобразовывать код, может быть, и не надо).

Виртуальная память не имеет указанных недостатков, но ее ключевой проблемой является преобразование виртуальных адресов в физические (почему это проблема, будет ясно дальше, а пока можно отметить существенные затраты времени на этот процесс, если не принять специальных мер).

**6.2. Функции ОС по управлению памятью**

Под памятью (memory) в данном случае подразумевается оперативная (основная) память компьютера. В однопрограммных операционных 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 многопрограммных ОС "пользовательская" часть памяти – важнейший ресурс вычислительной системы – должна быть распределена для размещения нескольких процессов, в том числе процессов ОС. Эта задача распределения выполняется операционной системой динамически специальной подсистемой управления памятью (*memory management*). Эффективное управление памятью жизненно важно для многозадачных систем. Если в памяти будет находиться небольшое число процессов, то значительную часть времени процессы будут находиться в состоянии ожидания ввода-вывода и загрузка процессора будет низкой.

В ранних ОС управление памятью сводилось просто к загрузке программы и ее данных из некоторого внешнего накопителя (перфоленты, магнитной ленты или магнитного диска) в ОЗУ. При этом память разделялась между программой и ОС. На рис. 6.3 показаны три варианта такой схемы. Первая модель раньше применялась на мэйнфреймах и мини-компьютерах. Вторая схема сейчас используется на некоторых карманных компьютерах и встроенных системах, третья модель была характерна для ранних персональных компьютеров с MS-DOS.



**Рис. 6.3.**  Варианты распределения памяти

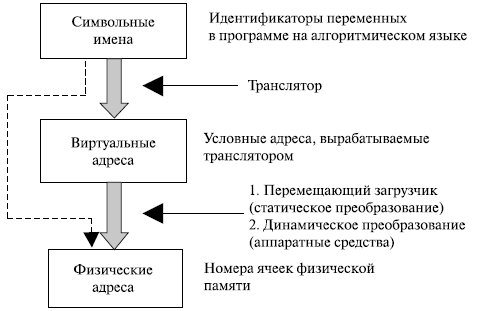
С появлением мультипрограммирования задачи ОС, связанные с распределением имеющейся памяти между несколькими одновременно выполняющимися программами, существенно усложнились.

Функциями ОС по управлению памятью в мультипрограммных системах являются:

* отслеживание (учет) свободной и занятой памяти;
* первоначальное и *динамическое выделение памяти* процессам приложений и самой операционной системе и освобождение памяти по завершении процессов;
* настройка адресов программы на конкретную область физической памяти;
* полное или частичное вытеснение кодов и данных процессов из ОП на диск, когда размеры ОП недостаточны для размещения всех процессов, и возвращение их в ОП;
* защита памяти, выделенной процессу, от возможных вмешательств со стороны других процессов;
* *дефрагментация памяти*.

Перечисленные функции особого пояснения не требуют, остановимся только на задаче преобразования адресов программы при ее загрузке в ОП.

Для идентификации переменных и команд на разных этапах жизненного цикла программы используются символьные имена, виртуальные (математические, условные, логические – все это синонимы) и физические адреса (рис. 6.4).



**Рис. 6.4.**  Типы адресов

Символьные имена присваивает пользователь при написании программ на алгоритмическом языке или ассемблере. Виртуальные адреса вырабатывает транслятор, переводящий программу на машинный язык. Поскольку во время трансляции неизвестно, в какое место оперативной памяти будет загружена программа, транслятор присваивает переменным и командам виртуальные (условные) адреса, считая по умолчанию, что начальным адресом программы будет нулевой адрес.

Физические адреса соответствуют номерам ячеек оперативной памяти, где в действительности будут расположены переменные и команды.

Совокупность виртуальных адресов процесса называется виртуальным адресным пространством. Диапазон адресов виртуального пространства у всех процессов один и тот же и определяется разрядностью адреса процессора (для Pentium адресное пространство составляет объем, равный 232 байт, с диапазоном адресов от 0000.000016 до FFFF.FFFF16).

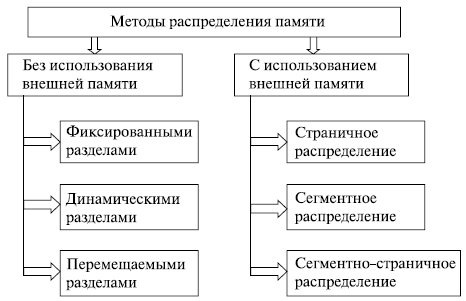
Существует два принципиально отличающихся подхода к преобразованию виртуальных адресов в физические. В первом случае такое преобразование выполняется один раз для каждого процесса во время начальной загрузки программы в память. Преобразование осуществляет перемещающий загрузчик на основании имеющихся у него данных о начальном адресе физической памяти, в которую предстоит загружать программу, а также информации, предоставляемой транслятором об адресно-зависимых элементах программы.

Второй способ заключается в том, что программа загружается в память в виртуальных адресах. Во время выполнения программы при каждом обращении к памяти операционная система преобразует виртуальные адреса в физические.

**6.3. Распределение памяти**

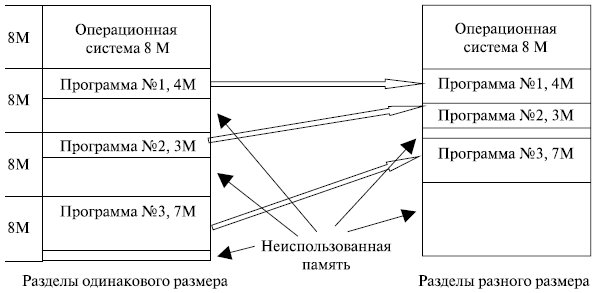
Существует ряд базовых вопросов управления памятью, которые в различных ОС решаются по-разному. Например, следует ли назначать каждому процессу одну непрерывную область физической памяти или можно выделять память участками? Должны ли сегменты программы, загруженные в память, находиться на одном месте в течение всего периода выполнения процесса или их можно время от времени сдвигать? Что делать, если сегменты программы не помещаются в имеющуюся память? Как сократить затраты ресурсов системы на управление памятью? Имеется и ряд других не менее интересных проблем управления памятью [5, 10, 13, 17].

Ниже приводится классификация методов распределения памяти, в которой выделено два класса методов – с перемещением сегментов процессов между ОП и ВП (диском) и без перемещения, т.е. без привлечения внешней памяти (рис. 6.5). Данная классификация учитывает только основные признаки методов. Для каждого метода может быть использовано несколько различных алгоритмов его реализации.



**Рис. 6.5.**  Классификация методов распределения памяти

На рис. 6.6 показаны два примера фиксированного распределения. Одна возможность состоит в использовании разделов одинакового размера. В этом случае любой процесс, размер которого не превышает размера раздела, может быть загружен в любой доступный раздел. Если все разделы заняты и нет ни одного процесса в состоянии готовности или работы, ОС может выгрузить процесс из любого раздела и загрузить другой процесс, обеспечивая тем самым процессор работой.



**Рис. 6.6.**  Варианты фиксированного распределения памяти

При использовании разделов с одинаковым размером имеются две проблемы.

1. Программа может быть слишком велика для размещения в разделе. В этом случае программист должен разрабатывать программу, использующую оверлеи, чтобы в любой момент времени требовался только один раздел памяти. Когда требуется модуль, отсутствующий в данный момент в ОП, пользовательская программа должна сама его загрузить в раздел памяти программы. Таким образом, в данном случае управление памятью во многом возлагается на программиста.
2. Использование ОП крайне неэффективно. Любая программа, независимо от ее размера, занимает раздел целиком. При этом могут оставаться неиспользованные участки памяти большого размера. Этот феномен появления неиспользованной памяти называется внутренней фрагментацией (internal fragmentation).

Бороться с этими трудностями (хотя и не устранить полностью) можно посредством использования разделов разных размеров. В этом случае программа размером до 8 Мбайт может обойтись без оверлеев, а разделы малого размера позволяют уменьшить внутреннюю фрагментацию при загрузке небольших программ.

В том случае, когда разделы имеют одинаковый раздел, размещение процессов тривиально – в любой свободный раздел. Если все разделы заняты процессами, которые не готовы к немедленной работе, любой из них может быть выгружен для освобождения памяти для нового процесса.

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 вместить данный процесс (в этом случае в задании пользователя указывался размер требуемой памяти). При таком подходе для каждого раздела требуется очередь планировщика, в которой хранятся выгруженные из памяти процессы, предназначенные для данного раздела памяти. Достоинство такого способа в возможности распределения процессов между разделами ОП так, чтобы минимизировать внутреннюю фрагментацию.

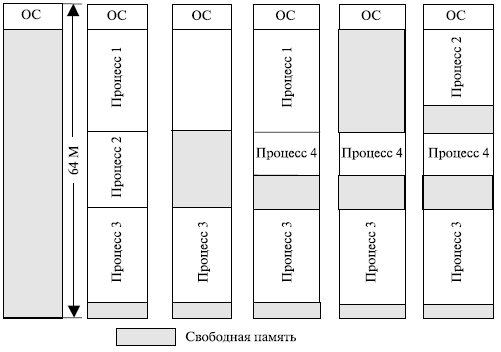
Недостаток заключается в том, что отдельные очереди для разделов могут привести к неоптимальному распределению памяти системы в целом. Например, если в некоторый момент времени нет ни одного процесса размером от 7 до 12 Мбайт, то раздел размером 12 Мбайт будет пустовать, в то время как он мог бы использоваться меньшими процессами. Поэтому более предпочтительным является использование одной очереди для всех процессов. В момент, когда требуется загрузить процесс в ОП, выбирается наименьший доступный раздел, способный вместить данный процесс.

В целом можно отметить, что схемы с *фиксированными разделами* относительно просты, предъявляют минимальные требования к операционной системе; накладные расходы работы процессора на распределение памяти невелики. Однако у этих схем имеются серьезные недостатки.

1. Количество разделов, определенное в момент генерации системы, ограничивает количество активных процессов (т.е. уровень мультипрограммирования).
2. Поскольку размеры разделов устанавливаются заранее во время генерации системы, небольшие задания приводят к неэффективному использованию памяти. В средах, где заранее известны потребности в памяти всех задач, применение рассмотренной схемы может быть оправдано, но в большинстве случаев эффективность этой технологии крайне низка.

Для преодоления сложностей, связанных с фиксированным распределением, был разработан альтернативный подход, известный как динамическое распределение. В свое время этот подход был применен фирмой IBM в операционной системе для мэйнфреймов в OS/MVT (мультипрограммирование с переменным числом задач –*Multiprogramming* With a Variable number of Tasks). Позже этот же подход к распределению памяти использован в ОС ЕС ЭВМ [12] .

При динамическом распределении образуется перемененное количество разделов переменной длины. При размещении процесса в основной памяти для него выделяется строго необходимое количество памяти. В качестве примера рассмотрим использование 64 Мбайт (рис. 6.7) основной памяти. Изначально вся память пуста, за исключением области, задействованной ОС. Первые три процесса загружаются в память, начиная с адреса, где заканчивается ОС, и используют столько памяти, сколько требуется данному процессу. После этого в конце ОП остается свободный участок памяти, слишком малый для размещения четвертого процесса. В некоторый момент времени все процессы в памяти оказываются неактивными, и операционная система выгружает второй процесс, после чего остается достаточно памяти для загрузки нового, четвертого процесса.



**Рис. 6.7.**  Вариант использования памяти

Поскольку процесс 4 меньше процесса 2, появляется еще свободный участок памяти. После того как в некоторый момент времени все процессы оказались неактивными, но стал готовым к работе процесс 2, свободного места в памяти для него не находится, а ОС вынуждена выгрузить процесс 1, чтобы освободить необходимое место и разместить процесс 2 в ОП. Как показывает данный пример, этот метод хорошо начинает работу, но плохо продолжает. В конечном счете, он приводит к наличию множества мелких свободных участков памяти, в которых нет возможности разместить какой-либо новый процесс. Это явление называется внешней фрагментацией (external fragmentation), что отражает тот факт, что сильно фрагментированной становится память, внешняя по отношению ко всем разделам.

Один из методов преодоления внешней фрагментации – уплотнение (compaction) процессов в ОП. Осуществляется это перемещением всех занятых участков так, чтобы вся свободная память образовала единую свободную область. В дополнение к функциям, которые ОС выполняет при распределении памяти динамическими разделами, в данном случае она должна еще время от времени копировать содержимое разделов из одного места в другое, корректируя таблицы свободных и занятых областей. Эта процедура называется уплотнением или сжатием.

Перечислим функции операционной системы по управлению памятью в этом случае.

1. Перемещение всех занятых участков в сторону старших или младших адресов при каждом завершении процесса или для вновь создаваемого процесса в случае отсутствия раздела достаточного размера.
2. Коррекция таблиц свободных и занятых областей.
3. Изменение адресов команд и данных, к которым обращаются процессы при их перемещении в памяти, за счет использования *относительной адресации*.
4. Аппаратная поддержка процесса динамического преобразования относительных адресов в абсолютные адреса основной памяти.
5. Защита памяти, выделяемой процессу, от взаимного влияния других процессов.

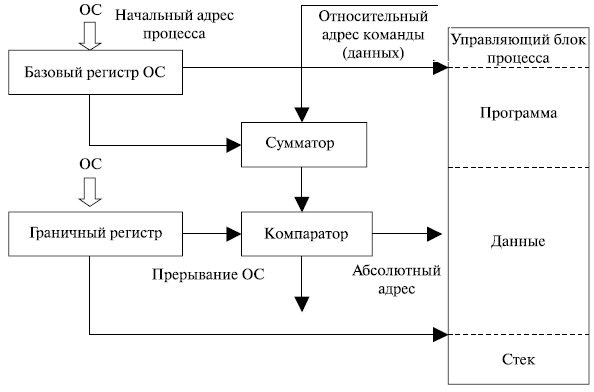
Уплотнение может выполняться либо при каждом завершении процесса, либо только тогда, когда для вновь создаваемого процесса нет свободного раздела достаточного размера. В первом случае требуется меньше вычислительной работы при корректировке таблиц свободных и занятых областей, а во втором – реже выполняется процедура сжатия.

Так как программа перемещается по оперативной памяти в ходе своего выполнения, в данном случае невозможно выполнить настройку адресов с помощью перемещающего загрузчика. Здесь более подходящим оказывается динамическое преобразование адресов. Достоинствами распределения памяти перемещаемыми разделами являются эффективное использование оперативной памяти, исключение внутренней и внешней фрагментации, недостатком – дополнительные накладные расходы ОС.

При использовании фиксированной схемы распределения процесс всегда будет назначаться одному и тому же разделу памяти после его выгрузки и последующей загрузке в память. Это позволяет применять простейший загрузчик, который замещает при загрузке процесса все относительные ссылки абсолютными адресами памяти, определенными на основе базового адреса загруженного процесса.

Ситуация усложняется, если размеры разделов равны (или неравны) и существует единая очередь процессов, – процесс по ходу работы может занимать разные разделы. Такая же ситуация возможна и при динамическом распределении. В этих случаях расположение команд и данных, к которым обращается процесс, не является фиксированным и изменяется всякий раз при выгрузке, загрузке или перемещении процесса. Для решения этой проблемы в программах используются относительные адреса. Это означает, что все ссылки на память в загружаемом процессе даются относительно начала этой программы. Таким образом, для корректной работы программы требуется аппаратный механизм, который бы транслировал относительные адреса в физические в процессе выполнения команды, обращающейся к памяти.

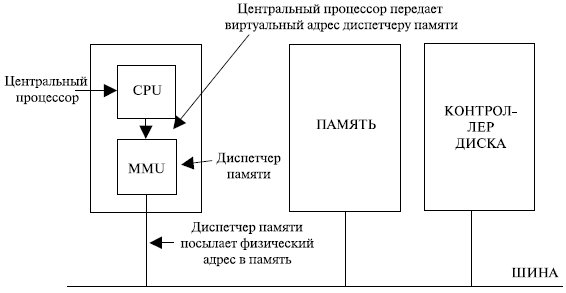
Применяемый обычно способ трансляции показан на рис. 6.8. Когда процесс переходит в состояние выполнения, в специальный регистр процесса, называемый базовым, загружается начальный адрес процесса в основной памяти. Кроме того, используется "граничный" (bounds) регистр, в котором содержится адрес последней ячейки программы. Эти значения заносятся в регистры при загрузке программы в основную память. При выполнении процесса относительные адреса в командах 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.



**6.4. Страничная организация виртуальной памяти**

Большинство систем виртуальной памяти используют технику, называемую страничной организацией памяти [32, 37]. Любой процесс, реализуемый в компьютере, может обратиться к множеству адресов в памяти. Адреса могут формироваться с применением индексации, базовых регистров, сегментных регистров и другими путями. 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 показано на рис. 6.9. Здесь диспетчер памяти показан как часть микросхемы процессора, как обычно и бывает чаще всего. Но логически он мог бы быть отдельной микросхемой, как было в недавнем прошлом.



**Рис. 6.9.**  Диспетчер памяти

Все имеющееся в настоящее время множество реализаций виртуальной памяти различается в основном способом структуризации виртуального адресного пространства.

Сам термин "виртуальная память" ассоциируется с системами, использующими страничную организацию. Впервые сообщение о виртуальной памяти на основе страничной организации появилось в 1962 году в работе Kilburn I и др. "One-Level *Storage System*", и вскоре после этого виртуальная память стала широко применяться в коммерческих системах.

В настоящее время выделяют три метода реализации виртуальной памяти.

1. *Страничная виртуальная память* организует перемещение данных между основной памятью и диском страницами – частями виртуального адресного пространства фиксированного и сравнительно небольшого размера.
2. Сегментная виртуальная память предусматривает перемещение данных сегментами – частями виртуального адресного пространства произвольного размера, полученного с учетом смыслового значения данных.
3. Сегментно-*страничная виртуальная память* использует двухуровневое деление: виртуальное адресное пространство делится на сегменты, а затем сегменты делятся на страницы. Единицей перемещения данных является страница.

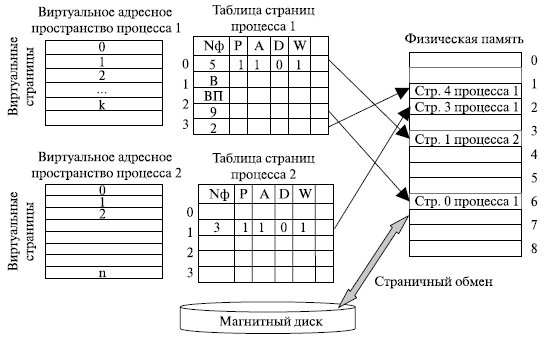
Для временного хранения сегментов и страниц на диске отводится специальная область либо специальный файл (страничный файл или файл подкачки – paging file). Текущий размер страничного файла является важным параметром, оказывающим влияние на возможности операционной системы: чем больше страничный файл, тем больше приложений может одновременно выполнять ОС (при фиксированном размере оперативной памяти). Однако необходимо понимать, что увеличение числа одновременно работающих приложений за счет увеличения размера страничного файла замедляет их работу, так как значительная часть времени при этом тратится на перемещение данных на диск и обратно.

Размер страничного файла в современных ОС является настраиваемым параметром, который выбирается администратором системы для достижения компромисса между уровнем программирования и быстродействием системы.

При страничной организации виртуальное адресное пространство каждого процесса делится на части одинакового, фиксированного для данной системы размера, называемые *виртуальными страницами* (Virtual pages). В общем случае размер виртуального адресного пространства не кратен размеру страницы, поэтому последняя страница дополняется фиксированной областью.

Вся оперативная память машины также делится на части такого же размера, называемые физическими страницами (или блоками, или кадрами). Размер страницы выбирается равным степени двойки: 1024, 2048, 4096 байт и т.д. Это позволяет упростить механизм преобразования адресов.

При создании процесса ОС загружает в операционную память несколько его *виртуальных страниц* (начальные страницы кодового сегмента и сегмента данных). Копия всего виртуального адресного пространства процесса находится на диске. Смежные виртуальные страницы не обязательно находятся в смежных физических страницах. Для каждого процесса ОС создает таблицу страниц – информационную структуру, содержащую записи обо всех виртуальных страницах процесса (рис. 6.10).



**Рис. 6.10.**  Таблицы страниц виртуальной памяти

Запись таблицы (дескриптор страницы) включает следующую информацию:

1. номер физической страницы (N ф.с.), в которую загружена данная *виртуальная страница*;
2. признак присутствия Р, устанавливаемый в единицу, если данная страница находится в оперативной памяти;
3. признак модификации страницы D, который устанавливается в единицу всякий раз, когда производится запись по адресу, относящемуся к данной странице;
4. признак обращения А к странице, называемый также битом доступа, который устанавливается в единицу при каждом обращении по адресу, относящемуся к данной странице;
5. другие управляющие биты, служащие, например, для целей защиты или совместного использования памяти на уровне страниц.

Перечисленные признаки в большинстве моделей процессов устанавливаются аппаратно схемами процессора при выполнении операций с памятью. Информация из таблицы страниц используется для решения вопроса о необходимости перемещения той или иной страницы между памятью и диском, а также для преобразования виртуального адреса в физический. Сами таблицы страниц, так же как и описываемые ими страницы, размещаются в оперативной памяти.

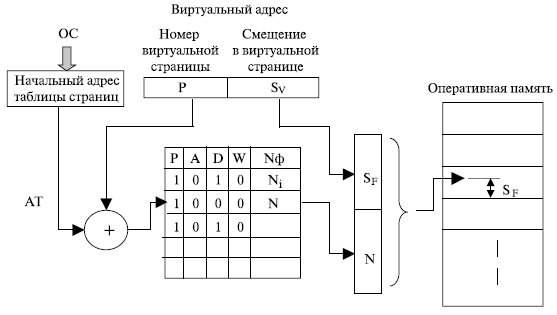
Поскольку процесс может задействовать большой объем виртуальной памяти (например, в Windows 2000 он равен 232 = 4 Гбайт), при использовании страницы объемом 4 Кбайт (212) потребуется 220 записей в таблице страниц для каждого процесса. Понятно, что выделять такое количество оперативной памяти под таблицы страниц нецелесообразно. Для преодоления этой проблемы большинство схем виртуальной памяти хранит таблицы страниц не в реальной, а в виртуальной памяти. Это означает, что сами таблицы страниц становятся объектами страничной организации. При работе процесса как минимум часть его таблицы страниц должна располагаться в основной памяти, в том числе запись о странице, выполняющейся в настоящий момент. Адрес таблицы страниц включается в *контекст процесса*. При активизации очередного процесса ОС загружает адрес его таблицы страниц в специальный регистр.

При каждом обращении к памяти выполняется поиск номера *виртуальной страницы*, содержащей требуемый адрес, затем по этому номеру определяется нужный элемент таблицы страниц и из него извлекается описывающая страницу информация. Далее анализируется признак присутствия, и если данная *виртуальная страница* находится в оперативной памяти, то выполняется преобразование виртуального адреса в физический. Если же нужная *виртуальная страница* в данный момент выгружена на диск, то происходит страничное прерывание.

Выполняющий процесс переводится в состояние ожидания, активизируя процесс из очереди процессов, находящихся в состоянии готовности. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу и пытается ее загрузить в оперативную память. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно. Если же свободных страниц нет, то на основании принятой в данной системе*стратегии замещения* страниц решается вопрос о том, какую страницу следует выгрузить из оперативной памяти.

После того как выбрана страница, которая должна покинуть оперативную память, обнуляется ее *бит присутствия* и анализируется ее признак модификации. Если удаляемая страница за время последнего требования в оперативной памяти была модифицирована, то ее новая версия должна быть переписана на диск. Если нет, то принимается во внимание, что на диске уже имеется предыдущая копия этой *виртуальной страницы*, и никакой записи на диск не производится. Физическая страница объявляется свободной. Из соображений безопасности в некоторых системах освобождаемая страница обнуляется, чтобы невозможно было использовать содержимое выгруженной страницы. Для хранения информации о положении вытесненной страницы в страничном файле ОС может задействовать специальные поля таблицы страниц.

Виртуальный адрес при страничном распределении может быть представлен в виде пары ( P, Sv ), где Р – номер *виртуальной страницы* процесса (нумерация страниц начинается с 0), а Sv – смещение в пределах *виртуальной страницы* (рис. 6.11). Физический адрес также может быть представлен в виде пары ( N, Sf ), где N – номер физической страницы, а Sf – смещение а пределах физической страницы. Задача подсистемы виртуальной памяти состоит в отображении пары значений ( P, Sv ) в пару ( N, Sf ).



**Рис. 6.11.**  Преобразование виртуального адреса

Чтобы понять механизм реализации этого отображения, следует остановиться на двух базисных свойствах страничной организации. Как уже отмечалось, объем страницы, как виртуальной, так и физической, выбирается равным степени двойки – 2к ( k = 8 и более). Отсюда следует, что смещение Sv и Sf может быть получено отделением k младших разделов в двоичной записи виртуального и, соответственно, физического адреса страницы. При этом оставшиеся старшие разделы адреса представляют собой двоичную запись номера виртуальной и, соответственно, физической страницы. Дополнив эти номера к нулям, можно получить начальный адрес виртуальной и физической страниц.

Второе свойство – линейность адресного пространства виртуальной и физической страницы – приводит к тому, что Sf = Sv. Отсюда следует простая схема преобразования виртуального адреса в физический.

При обращении к памяти по некоторому виртуальному адресу ( P, Sv ) аппаратные схемы процессора выполняют следующие действия.

1. Из специального регистра процессора извлекается начальный адрес АТ таблицы страниц активного процесса. С помощью сумматора http://www.intuit.ru/img/tex/0a9ca579028b26ad41e966040a66325a.png по значениям АТ, Р, L (длина отдельной записи в таблице страниц) определяется адрес нужной записи в таблице страниц:

A = АТ + (Р \* L).

1. Считывается номер соответствующей физической страницы – N.
2. К номеру физической страницы присоединяется смещение Sv.

В итоге полученный физический адрес оперативной памяти представляется парой значений ( N, Sf ).

Рассмотрим пример, поясняющий основные характеристики организации страничной виртуальной памяти. Пусть компьютер имеет оперативную память объемом Еоп= 256 Мбайт, размер страницы выбран равным Естр= 4 Кбайт. В этом случае количество физических страниц равно

Nf = Еоп / Естр = 256\*2020 / 4\*210 = 64.000 страниц.

Для отображения физического адреса произвольного байта оперативной памяти потребуется K = log2 256\*2020 = 28 двоичных разрядов.

Число разрядов для отображения смещения в странице M = log2 4 Кбайт = log2 4096 = 12.

Если процессор имеет 32-разрядную структуру, то на номер *виртуальной страницы* отводится 32-12=20 двоичных разрядов. Таким образом, число *виртуальных страниц* равно Nв = 220 (примерно 1 млн *виртуальных страниц*).

Для каждой *виртуальной страницы* в таблице страниц должна быть запись, содержащая номер *виртуальной страницы* (20 двоичных разрядов), начальный адрес соответствующей ей физической страницы плюс дополнительные разряды, характеризующие свойства страницы (присутствие, модификация, обращение и т.п.), на которые потребуется 1 байт. Поскольку адрес начала физической страницы кратен 4096, то на него достаточно 28 – 12 = 16 двоичных разрядов (остальные 12 разрядов заполняются нулями). Таким образом, одна запись таблицы страниц займет 20 + 16 + 8 = 44 двоичных разрядов или 6 байт. Общий объем таблицы страниц составит 6 \* Nв = 6 Мбайт.

Реально при выборе структуры записи таблицы страниц нужно учитывать следующие факторы. Современные компьютеры позволяют наращивать объем оперативной памяти (например, в ПК она может почти достигать объема виртуальной памяти и даже более). Поэтому на адрес физической страницы в нашем примере следует выделить 32-12 = 20 двоичных разрядов. С другой стороны, нет необходимости в записи (дескрипторе) *виртуальной страницы* иметь поле с номером *виртуальной страницы* (20 разрядов), так как адрес нужной записи можно вычислять, как это было рассмотрено выше. Следовательно, в нашем примере длина записи должна быть равной 32 - 12 + 8 = 28 двоичным разрядам, т.е. с округлением до целого числа байт – 4 байт. Таким образом, для каждого выполняющегося в компьютере процесса ОС должна создать страничную таблицу размером 4 \* Nв байт = 5 \* 220 = 4 Мбайт.

Процедура преобразования виртуального адреса в физический без принятия специальных мер (кэширование активных страниц) занимает один цикл оперативной памяти, который затрачивается на считывание номера физической страницы из таблицы страниц. Поэтому любое обращение к ОП будет занимать 2 цикла вместо одного при работе без виртуальной памяти. Другим фактором, влияющим на производительность систем, являются затраты времени на обработку страничных прерываний. При неправильно выбранной *стратегии замещения* страниц может возникнуть ситуация, когда система тратит большую часть времени впустую на подкачку страниц из оперативной памяти на диск и обратно.

**6.5. Оптимизация функционирования страничной виртуальной памяти**

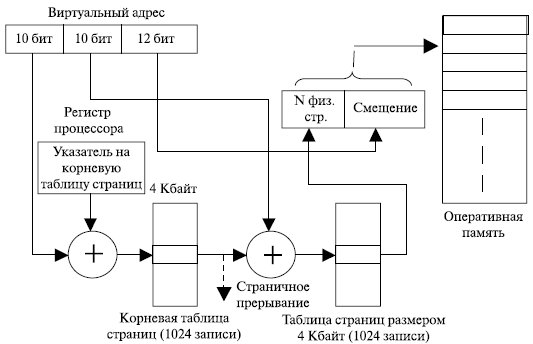
В настоящее время известно несколько методов повышения эффективности функционирования страничной виртуальной памяти. К ним относятся [17]:

1. более сложная структуризация виртуального адресного пространства, например, двухуровневая (типичная для 32-битовой адресации);
2. использование специального высокоскоростного кэша для хранения части записей таблицы страниц, который обычно называют буфером быстрого преобразования адреса, или буфером поиска трансляции (translation lookaside buffer – TLB);
3. выбор оптимального размера страницы виртуальной памяти;
4. эффективное управление страничным обменом.

Остановимся на возможностях реализации этих методов.

Рассмотрим вариант двухуровневой страничной организации. При такой схеме имеется каталог таблиц страниц, в котором каждая запись указывает на таблицу страниц. Таким образом, если размер каталога – X, а максимальный размер таблицы страниц – Y, то процесс может состоять максимум из X и Y страниц. Обычно максимальный размер таблицы страниц определяется условием ее размещения на одной странице (такой подход используется в процессоре Pentium).

На рис. 6.12 приведен пример двухуровневой схемы, типичной для 32-битовой адресации. Подобная схема позволяет существенно сохранить размер пользовательской таблицы страниц, размещаемой в основной памяти (с 4 Мбайт до 4 Кбайт). В данном случае виртуальное адресное пространство пользовательского процесса может составлять 232 = 4 Гбайт. При объеме страницы 212 = 4 Кбайт в этом пространстве размещается 232 / 212 = 220 страниц. Таким образом, пользовательская таблица страниц будет иметь 220 4-байтных записей общим объемом 4 Мбайт. Разместить такие таблицы для нескольких процессов в ОП нереально. В двухуровневой схеме это и не требуется. В основной памяти постоянно находится корневая таблица, содержащая 1024 записей, указывающих на начальный адрес пользовательской таблицы страниц (ее объем, как указано выше, 4 Мбайт). Указание на начальный адрес корневой таблицы (активного процесса) заносится в регистр процессора. Первые 10 бит виртуального адреса используются для индексации в корневой таблице для поиска записей о странице пользовательской таблицы.



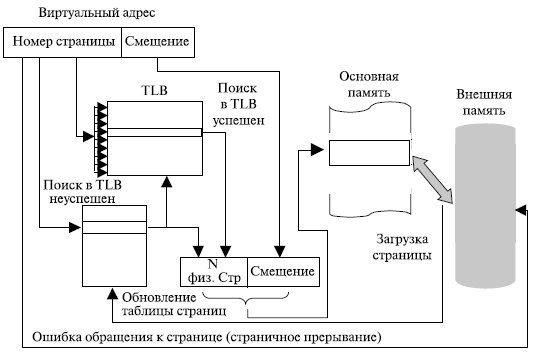
**Рис. 6.12.**  Двухзвенная схема таблиц страниц

Если страница находится в ОП, то следующие 12 бит виртуального адреса используются для задания смещения в физической странице ОП. В противном случае генерируется страничное прерывание, но уже из-за отсутствия нужной страницы процесса в ОП.

Таким образом, двухуровневая схема, сокращая объем памяти для хранения таблицы страниц, в общем случае замедляет преобразование виртуального адреса за счет большего числа возможных страничных прерываний (даже если нет страничного прерывания, требуется три цикла ОП вместо двух при одноуровневой страничной организации).

Как уже отмечалось, простая схема страничной виртуальной памяти, по сути, удваивает время обращения к памяти. Для преодоления этой проблемы большинство реально применяющихся схем виртуальной памяти используют специальный высокоскоростной кэш для записей таблицы страниц, который обычно называют буфером быстрой трансляции адресов – TLB. Этот кэш функционирует так же как и обычный кэш памяти и содержит те записи таблицы страниц, которые использовались последними. Организация аппаратной поддержки использования TLB показана на рис. 6.13.

Получив виртуальный адрес, процессор просматривает TLB. Если требуемая запись найдена, процессор получает адрес физической страницы и формирует реальный адрес. Если запись в TLB не найдена, то процессор берет номер *виртуальной страницы* в качестве индекса для таблицы страниц процесса и просматривает соответствующую запись. Если *бит присутствия* в ней установлен, значит, искомая страница находится в основной памяти, и процессор получает номер физической страницы из записи таблицы страниц, а затем формирует реальный физический адрес. Одновременно вносится использованная запись таблицы страниц в TLB.

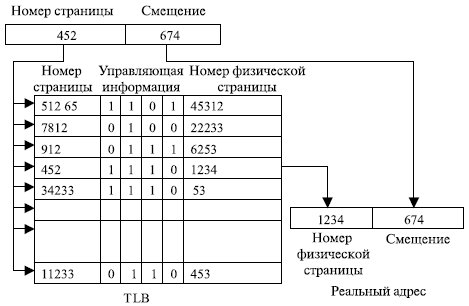


**Рис. 6.13.**  Буфер быстрой переадресации

Если *бит присутствия* данной *виртуальной страницы* не установлен, это означает, что искомой страницы в оперативной памяти нет. В этом случае процессор генерирует сигнал страничного прерывания, активизирующий операционную систему, которая загружает требуемую страницу в оперативную память и обновляет таблицу страниц.

Практика использования виртуальной памяти показала, что для нее справедлив закон локализации большинства обращений в небольшое количество недавно использованных страниц. При этом соответствующие записи будут находиться в кэше, так что с помощью TLB существенно повышается эффективность работы виртуальной памяти.

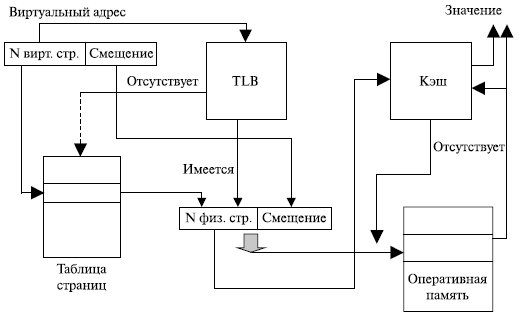
В организации TLB имеется ряд особенностей. Поскольку TLB содержит только некоторые из записей таблицы страниц (32 в процессоре Pentium), индексация записей в TLB на основе номера страницы не представляется возможной. Вместо этого каждая запись TLB должна наряду с полной информацией из записи таблицы страниц включать также номер *виртуальной страницы*. Процессор аппаратно способен одновременно опрашивать все записи TLB для определения того, какая из них соответствует заданному номеру страницы. Такой подход известен как ассоциативное отображение (associative mapping), в отличие от прямого отображения, или индексирования, применяемого для поиска в таблице страниц, как показано на рис. 6.14.



**Рис. 6.14.**  Ассоциативная память

Конструкция TLB должна также предусматривать способы организации записей в кэш и принятия решения о том, какая из старых записей должна быть удалена при внесении в кэш новой записи.

Следует подчеркнуть, что механизм виртуальной памяти должен взаимодействовать с кэшем оперативной памяти (кроме TLB). Это взаимодействие показано на рис. 6.15. Сначала происходит обращение к TLB для выяснения наличия в нем соответствующей записи таблицы страниц. При положительном результате путем объединения номера физической страницы, получаемого из TLB, и смещения генерируется физический адрес. Если требуемой записи в TLB нет, она выбирается из таблицы страниц. После получения физического адреса в обеих ситуациях выполняется обращение к кэшу для выяснения наличия в нем блока с требуемым физическим адресом. Если ответ положительный, то требуемое значение (код или данные) передается процессору. В противном случае производится выборка слова из основной памяти и обновляется содержимое кэша основной памяти.

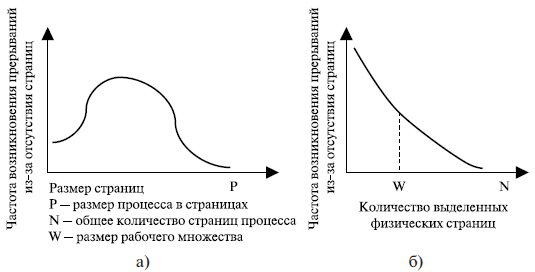


**Рис. 6.15.**  Использование кэша оперативной памяти

При выборе размера страницы нужно учитывать несколько факторов. Один из них – внутренняя фрагментация, которая напрямую зависит от размера страницы. Внутренняя фрагментация уменьшается с уменьшением размера страницы. Однако, чем меньше страницы, тем больше их требуется для процесса, что означает увеличение размера таблицы страниц. При этом для больших программ в загруженной многозадачной среде это приведет к тому, что часть страничных таблиц активных процессов будет находиться в виртуальной памяти, и при отсутствии страницы будет возникать двойное прерывание: первое – для получения требуемой записи из таблицы страниц, второе – для получения доступа к требуемой странице процесса.

Такое двойное прерывание существенно снизит производительность виртуальной памяти. Кроме того, следует учитывать и факт повышения скорости работы диска при передаче больших блоков данных. Таким образом, страницы небольшого размера нецелесообразны, поскольку уменьшение внутренней фрагментации в этом случае не столь значительно по сравнению со снижением производительности виртуальной памяти.

Проблема выбора размера страницы усложняется еще и тем, что размер страницы влияет и на частоту возникновения прерывания из-за отсутствия страницы в основной памяти. На рис. 6.16 а) показан примерный график изменения частоты страничных прерываний из-за отсутствия страницы с учетом принципа локализации. Если размер страницы очень мал, в памяти размещается относительно большое число страниц процесса. Через некоторое время страницы в памяти будут содержать части процесса, сосредоточенные вблизи последних обращений, и частота прерываний из-за отсутствия страницы должна быть невелика.



**Рис. 6.16.**  Изменение частоты страничных прерываний

По мере увеличения размера страницы каждая отдельная страница будет содержать данные, которые располагаются все дальше и дальше от последних выполненных обращений к памяти. Действие принципа локализации ослабевает, и наблюдается рост количества прерываний из-за отсутствия страницы. С дальнейшим ростом размера страницы он (размер) становится сравнимым с размером процесса (точка Р на графике) и прерывания становятся реже, а достижения размера этого процесса прекращаются.

Следует учитывать также влияние количества физических страниц, распределенных процессу. На рис. 6.16 б) показано, что для фиксированного размера страницы частота прерываний из-за отсутствия страницы уменьшается с ростом числа страниц, находящихся в основной памяти.

Реально размеры страниц различных компьютеров составляют следующие значения: 512 байт (семейство VAX, IBM AS/400), 4 Кбайт (IBM 370, MIPS), 8 Кбайт (DEC Alpha), от 4 Кбайт до 4 Мбайт (Pentium).

Решение об используемом размере страниц связано также с размером физической памяти и размером программы. Нужно также учитывать тот факт, что современные технологии программирования приводят к снижению локализации ссылок процесса. Например, объектно-ориентированные технологии стимулируют применение множества мелких модулей кода и данных с обращениями к большому количеству объектов за относительно короткое время (если программа на языке С для небольших задач занимает 3 – 4 Кбайт, то та же программа на Visual С++ займет сотни Кбайт). Многопоточные приложения приводят к внезапным изменениям в потоке команд и обращениям к памяти, разбросанным по сильно отличающимся адресам.

Новые тенденции в программировании приводят к тому, что снижается результативность поиска в TLB с ростом размеров процесса и уменьшением локализации обращений в программе. Таким образом, TLB может стать узким местом, ограничивающим производительность виртуальной памяти. Чтобы повысить производительность TLB, нужно увеличить его емкость. Однако увеличение размера TLB связано с другими аспектами аппаратного решения вопросов обращения к памяти – такими как размер кэша основной памяти и количество обращений к памяти при выполнении одной команды. Это приводит к выводу о невозможности роста размера TLB такими же темпами, как увеличение размеров памяти. Альтернативой может быть использование больших размеров страниц (в этом случае размер TLB может быть меньше, а TLB ссылается на большой блок данных). Однако в этом случае кэш ОП должен тоже быть большим. Кроме того, большие размеры страниц приведут к значительной внутренней фрагментации.

Учитывая все эти обстоятельства, в рядах компьютеров применяются множественные размеры страниц (что, однако, весьма сложно как в аппаратном аспекте, так и в программном в части операционной системы). Множественные размеры страниц обеспечивают гибкость, необходимую для использования TLB. Большие непрерывные области адресного пространства процесса, например программный код, могут отображаться с использованием небольшого количества страниц, в то время как стеки потоков могут использоваться для отображения страницы малого размера.

Одна из основных задач ОС – управление виртуальной памятью. При выборе стратегии решения этой задачи ключевым вопросом становится производительность: требуется сократить количество прерываний из-за отсутствия страницы в основной памяти, поскольку их обработка приводит к существенным накладным расходам. Кроме того, ОС должна активизировать готовый к работе процесс на время выполнения медленных операций ввода-вывода.

Для управления страничным обменом нужно решить следующие задачи [10, 17]:

* когда передавать страницу в основную память;
* где размещать страницу в физической памяти;
* какую страницу основной памяти выбрать для замедления, если в основной памяти нет свободной физической страницы;
* сколько страниц процесса следует загрузить в основную память;
* когда измененная страница, должна быть записана во вторичную память;
* сколько процессов размещать в основной памяти.

В соответствии с этими задачами ниже перечислены стратегии ОС для управления виртуальной памятью.

|  |  |
| --- | --- |
| **Наименование** | **Возможные алгоритмы (решения)** |
| *Стратегия выборки* (когда?) | По требованию, предварительная выборка |
| *Стратегия размещения* (где?) | Первый подходящий раздел (для сегментной виртуальной памяти). Любая страница физической памяти (для сегментно-страничной и страничной организации виртуальной памяти) |
| Стратегия замещения (какие?) | Оптимальный выбор, дольше всех неиспользовавшиеся. Первым вошел – первым вышел (FIFO), часовой, буферизация страниц |
| Управление резидентным множеством (сколько?) | Фиксированный размер, переменный размер, локальная и глобальная области видимости |
| Стратегия очистки (когда?) | По требованию, предварительная очистка |
| Управление загрузкой (сколько?) | Рабочее множество, критерии L=S и 50% |

*Стратегия выборки* определяется, когда страница должна быть передана в основную память. Два основных варианта – по требованию и предварительно. В первом случае страница передается в основную память только тогда, когда выполняется обращение к ячейке памяти, расположенной на этой странице. Если все прочие элементы системы управления памятью работают хорошо, то происходит следующее. Когда процесс только запускается, возникает поток прерываний обращений к страницам, но далее срабатывает принцип локализации, все большее количество обращений выполняется к недавно загруженным страницам, и количество прерываний существенно снижается.

В случае предварительной выборки загружается несколько страниц. Такая выборка использует особенности работы дисковых устройств, заключающиеся в том, что несколько последовательно расположенных страниц загрузятся значительно быстрее, чем загрузка этих же страниц по одной в течение некоторого промежутка времени.

Предварительная выборка планируется программистом при разработке программы. Тем не менее, выгодность предварительной выборки не доказана.

*Стратегия размещения* определяет, где именно в физической памяти будут располагаться части процесса. Для систем, использующих сегментно-страничную или чисто страничную организацию виртуальной памяти, *стратегия размещения* не актуальна, поскольку применение TLB и аппаратное обеспечение к памяти одинаково результативно при любых сочетаниях адресов виртуальных и физических страниц.

В многопроцессорных системах с *неоднородным доступом к памяти* (различные расстояния между процессорами и модулями памяти) *стратегия размещения*становится очень важной и требует всестороннего исследования.

Стратегия замещения определяет выбор страниц в основной памяти для замещения их загружаемыми из вторичной памяти страницами. Эта стратегия связана с решением следующих вопросов:

* какое количество страниц в основной памяти должно быть выделено каждому активному процессу;
* должны ли замещаемые страницы относиться к одному процессу или в качестве кандидатов на замещение должны рассматриваться все страницы оперативной памяти;
* какие именно страницы из рассматриваемого множества следует выбрать для замещения.

Первые два вопроса относятся к стратегии управления резидентным множеством, их рассмотрим далее. Третий вопрос напрямую связан со *стратегией замещения*. Все используемые *стратегии замещения* направлены на то, чтобы выгрузить страницу, обращений к которой в ближайшем будущем не последует. Большинство стратегий замещения пытаются определить будущее поведение программы на основе ее прошлого поведения. Независимо от стратегий управления резидентным множеством имеется ряд основных алгоритмов, применяемых для выбора замещаемой страницы:

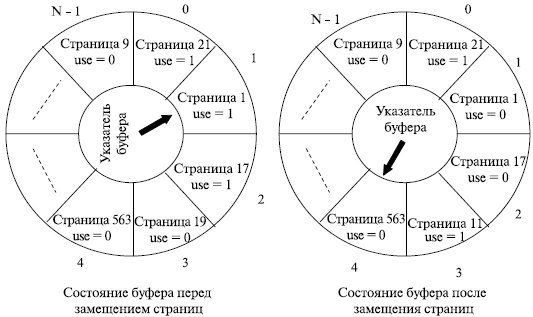
* оптимальный алгоритм;
* алгоритм дольше всех не использующейся страницы;
* алгоритм "первым вошел – первым вышел" (FIFO);
* часовой алгоритм и др.

Оптимальный алгоритм состоит в выборе замещения той страницы, обращение к которой будет через наибольший промежуток времени по сравнению со всеми остальными страницами. Понятно, что реализовать такой алгоритм невозможно, поскольку для этого системе требуется знать все будущие события. Однако он является стандартом, с которым сравниваются все алгоритмы.

Алгоритм FIFO рассматривает физические страницы процесса как циклический буфер, с циклическим удалением страниц из него. Это один из простейших в реализации алгоритмов. Логика его работы заключается в том, что замещается страница, находящаяся в памяти дольше других. Однако далеко не всегда эта страница редко используется.

Хотя алгоритм дольше всех не использующейся страницы близок к оптимальному, он труден в реализации и приводит к значительным накладным расходам. Разработано достаточно много алгоритмов, основанных на данной стратегии, многие из них представляют собой варианты схемы, известной как часовая стратегия (clock policy).

В простейшей схеме часовой стратегии с каждой физической страницей связан один бит, который называется битом использования (рис. 6.17). Когда *виртуальная страница* загружается впервые в физическую страницу, бит использования переводится в 1. При последующих обращениях к странице, вызвавших прерывание из отсутствия страницы, этот бит устанавливается равным 1. При работе алгоритма замещения множество страниц, являющихся кандидатами на замещение (текущий процесс, локальная область видимости или глобальная область видимости**1)** )), рассматривается как циклический буфер, с которым связан указатель.



**Рис. 6.17.**  Часовая стратегия замещения

При замещении страницы указатель перемещается к следующему кадру в буфере. Когда наступает время замещения страницы, ОС сканирует буфер для поиска кадра, бит использования которого равен 0. При этом когда в процессе поиска встречается кадр с битом использования, равным 1, он сбрасывается в 0. Первый же встречный кадр с нулевым битом использования выбирается для замещения. Если все кадры имеют бит использования, равный 1, указатель совершает полный круг и возвращается к начальному положению, заменяя страницу в этом кадре. Буфер кадров страниц представлен в виде круга, напоминающего часы, откуда и произошло название стратегии.

На рис. 6.17 приведен простейший пример использования часовой стратегии. Для замещения доступны n-1 кадров основной памяти, представленных в виде циклического буфера. Непосредственно перед тем как заместить страницу в буфере загружаемой из вторичной памяти страницей 11, указатель буфера указывает на кадр 1, содержащий страницу 1. Приступаем к выполнению часового алгоритма. Поскольку бит использования страницы 17 в кадре 2 равен 1, эта страница не замещается. Бит ее использования сбрасывается, а указатель перемещается к следующему кадру 3. Здесь находится страница 19, бит использования которой равен 0. Эта страница выбирается для замещения. На ее место загружается страница 11, бит использования которой переводится в 1. Указатель переводится на следующий кадр 4, и на этом выполнение алгоритма завершается. Повысить эффективность часового алгоритма можно путем увеличения количества используемых при его работе битов [17].

**6.6. Сегментная организация виртуальной памяти**

При страничной организации виртуальное адресное пространство делится на равные части механически без учета смыслового значения данных. Для многих задач наличие двух и более отдельных виртуальных адресных пространств может оказаться намного лучше, чем одно.

Например, у компилятора есть много таблиц, которые формируются по мере трансляции, включая в себя [10]:

1. исходный текст, сохраненный для печати листинга;
2. символьную таблицу, содержащую имена и атрибуты переменных;
3. таблицу, содержащую константы;
4. дерево грамматического разбора, содержащее синтаксический анализ программы;
5. стек, используемый для процедурных вызовов внутри компилятора.

Во время компиляции каждая из первых четырех таблиц непрерывно растет. Последняя таблица при компиляции непредсказуемо увеличивается или уменьшается. В одномерной памяти эти пять таблиц должны размещаться в смежных частях виртуального адресного пространства, как показано на рис. 6.18. В одномерном адресном пространстве при росте таблиц одна может "упереться" в другую. Можно было бы программным путем забирать памяти у одних таблиц и передавать другим. Но такая работа аналогична управлению собственными оверлеями, что представляет собой неудобство и большую скучную (возможно, неоплачиваемую) работу.



**Рис. 6.18.**  Сложности размещения в одном виртуальном адресном пространстве

Необходим метод, освобождающий программиста от управления расширяющимися и сокращающимися таблицами тем же самым способом, которым виртуальная память устраняет беспокойство организации оверлейных программ. Простое и предельно общее решение заключается в том, чтобы обеспечить машину множеством полностью независимых адресных пространств, называемых сегментами.

Каждый сегмент содержит линейную последовательность адресов от 0 до некоторого максимума. Различные сегменты могут быть различной длины. Более того, длины сегментов могут изменяться во время выполнения. Поскольку каждый сегмент составляет отдельное адресное пространство, разные сегменты могут расти и сокращаться независимо друг от друга.

Чтобы определить адрес в такой сегментированной или двумерной памяти, программа должна указать адрес, состоящий из двух частей: номер сегмента и адрес внутри сегмента. Максимальный размер сегмента определяется разрядностью виртуального адреса, например, при 32-разрядном микропроцессоре он равен 232 = 4 Гбайт. При этом максимально возможное виртуальное адресное пространство представляет набор из N виртуальных сегментов (заметим, что общего для сегментов линейного виртуального адреса не существует).

Стоит подчеркнуть, что сегмент – это логический объект, о чем программист знает и поэтому использует его как логический объект.

Помимо простоты управления увеличивающимися или сокращающимися структурами данных, сегментированная память обладает и другими преимуществами.

К ним относятся:

* простота компоновки отдельно скомпилированных процедур (обращение к начальной точке процедуры осуществляется адресом вида (n,0), где n – номер сегмента);
* легкость обеспечения дифференцируемого доступа к различным частям программы (например, запретить обращаться для записи в сегмент программы);
* простота организации совместного использования фрагментов программ различными процессами, например, библиотеки совместного доступа могут быть оформлены в виде отдельного сегмента, который может быть включен в виртуальное адресное пространство нескольких процессов.

Сравнение страничной организации памяти и сегментации приведено ниже.

|  |  |  |
| --- | --- | --- |
| **Вопрос** | **Страничная** | **Сегментация** |
| Нужно ли программисту знать о том, что используется эта техника? | Нет | Да |
| Сколько в системе линейных адресных пространств? | Одно | Много |
| Может ли суммарное адресное пространство превышать размеры физической памяти? | Да | Да |
| Возможно ли разделение процедур и данных, а также раздельная защита для них? | Нет | Да |
| Легко ли размещаются таблицы с непостоянными размерами? | Нет | Да |
| Облегчен ли совместный доступ пользователей к процедурам? | Нет | Да |
| Зачем была придумана эта техника? | Чтобы получить большое линейное адресное пространство без затрат на физическую память | Для разбиения программ и данных на независимые адресные пространства, облегчения защиты и совместного доступа |

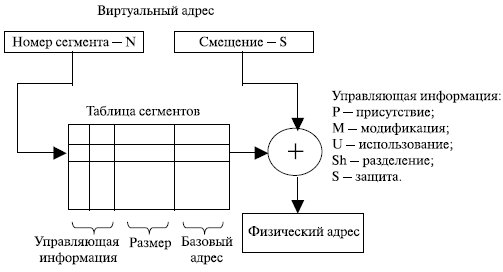
При загрузке процесса в оперативную память помещается только часть его сегментов, полная копия виртуального адресного пространства находится в дисковой памяти. Для каждого загружаемого сегмента ОС подыскивает непрерывный участок свободной памяти достаточного размера. Смежные в виртуальной памяти сегменты могут занимать несмежные участки оперативной памяти. Если во время выполнения процесса происходит обращение к отсутствующем в основной памяти сегменту, происходит прерывание. Операционная система в данном случае работает аналогично подобному процессу в страничной виртуальной памяти.

На этапе создания процесса во время загрузки его образа в оперативную память ОС создает таблицу сегментов процесса, аналогичную таблице страниц, в которой для каждого сегмента указывается:

* базовый физический адрес начала сегмента в оперативной памяти;
* размер сегмента;
* правила доступа к сегменту;
* признаки модификации, присутствия и обращения к данному сегменту, а также некоторая другая информация.

Если виртуальные адресные пространства нескольких процессов включают один и тот же сегмент, то в таблицах сегментов этих процессов делаются ссылки на один и тот же участок оперативной памяти, в который данный сегмент загружается в единственном экземпляре. Обычно программы в этих сегментах являются рентабельными (reentrant able), т.е. обладают свойством повторной входимости кода. Код таких программ не изменяется процессом.

Механизм преобразования виртуального адреса при сегментной организации очень схож с преобразованием виртуального адреса при страничной организации. Однако факт произвольного размера сегментов приводит к тому, что нельзя обойтись конкатенацией номера сегмента и смещения. В данном случае физический адрес получается сложением базового адреса сегмента, который определяется по номеру сегмента n из таблицы сегментов, и смещения S. Схема преобразования виртуального адреса при сегментной организации памяти приведена на рис. 6.19.

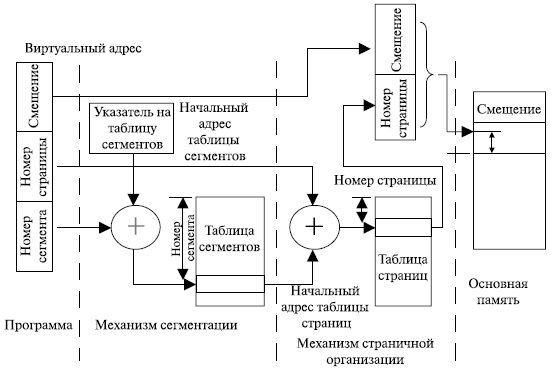


**Рис. 6.19.**  Преобразование виртуального адреса

Использование операции сложения вместо конкатенации замедляет процедуру преобразования виртуального адреса в физический. Другим недостатком сегментной организации виртуальной памяти является большая избыточность перемещения данных между диском и оперативной памятью, поскольку перемещаются целиком большие сегменты. Во многих случаях было бы достаточно загружать и выгружать не весь сегмент, а одну или несколько страниц. Однако наиболее существенный недостаток сегментной организации виртуальной памяти – внешняя фрагментация, которая возникает из-за произвольных размеров сегментов. Заметим, что внутренняя фрагментация, характерная для страничной организации виртуальной памяти, в данном случае отсутствует.

**6.7. Сегментно-страничная виртуальная память**

Данный метод организации виртуальной памяти направлен на сочетание достоинств страничного и сегментного методов управления памятью. В такой комбинированной системе адресное пространство пользователя разбивается на ряд сегментов по усмотрению программиста. Каждый сегмент в свою очередь разбивается на страницы фиксированного размера, равные странице физической памяти. С точки зрения программиста, логический адрес в этом случае состоит из номера сегмента и смещения в нем. С позиции операционной системы смещение в сегменте следует рассматривать как номер страницы определенного сегмента и смещение в ней (рис. 6.20).



**Рис. 6.20.**  Сегментно-страничная организация памяти

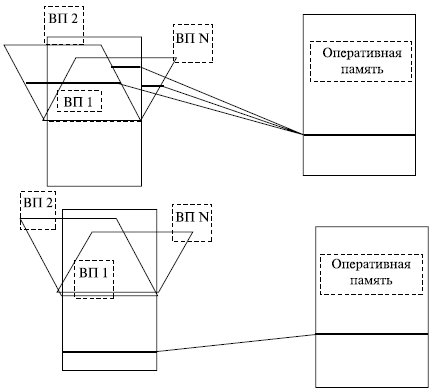
С каждым процессом связана одна таблица сегментов и несколько (по одной на сегмент) таблиц страниц. При работе определенного процесса в регистре процессора хранится начальный адрес соответствующей таблицы сегментов. Получив виртуальный адрес, процессор использует его часть, представляющую номер сегмента, в качестве индекса в таблице сегментов для поиска таблицы страниц данного сегмента. После этого часть адреса, представляющая собой номер страницы, используется для поиска номера физической страницы в таблице страниц. Затем часть адреса, представляющая смещения, используется для получения искомого физического адреса путем добавления к начальному адресу физической страницы.

Сегментация удобна для реализации защиты и совместного использования сегментов разными процессами. Поскольку каждая запись таблицы сегментов включает начальный адрес и значение длины, программа не в состоянии непреднамеренно обратиться к основной памяти за границами сегмента. Для того чтобы отличить разделяемые сегменты от индивидуальных, записи таблицы сегментов содержат 1-битовое поле, имеющее два значения: shared (разделяемый) или private (индивидуальный). Для осуществления совместного использования сегмента он помещается в виртуальное адресное пространство нескольких процессов, при этом параметры отображения этого сегмента настраиваются так, чтобы они соответствовали одной и той же области оперативной памяти (делается это указанием одного и того же базового физического адреса сегмента).

Возможен и более экономичный для ОС метод создания разделяемого виртуального сегмента – помещение его в общую часть виртуального адресного пространства, т.е. в ту часть, которая обычно задействуется для модулей ОС. В этом случае настройка соответствующей записи для разделяемого сегмента выполняется только один раз, а все процессы пользуются такой настройкой и совместно используют часть оперативной памяти.

Оба рассмотренных подхода к разделению сегмента можно иллюстрировать схемами, показанными ниже на рис. 6.21.

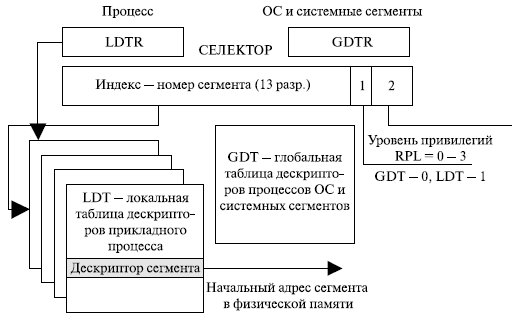
По второй схеме организована виртуальная память систем, работающих на процессоре Pentium. В Windows 2000 поддерживается 16 К независимых сегментов. У каждого процесса 4 Гбайт виртуального адресного пространства (из них 2 Гбайт отводится под ОС и 2 Гбайт – пользовательским программам). Основа виртуальной памяти Windows 2000 представляется двумя таблицами: локальной таблицей дескрипторов LDT (Local Descriptor Table) и глобальной таблицей дескрипторов GDT (Global Descriptor Table). У каждого процесса есть своя собственная таблица LDT, но глобальная таблица дескрипторов одна, ее совместно используют все процессы. Таблица LDT описывает сегменты, локальные для каждой программы, включая ее код, данные, стек и т.д.; таблица GDT несет информацию о системных сегментах, включая саму операционную систему.



**Рис. 6.21.**  Разделяемые сегменты

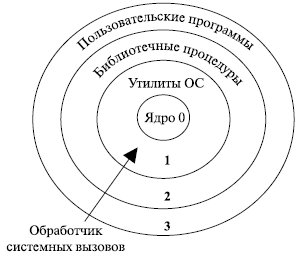
В каждый момент времени в специальных регистрах GDTR и LDTR хранится информация о местоположении и размерах глобальной таблицы GDT и активной таблице LDT. Регистр LDTR указывает на расположение сегмента LDT в оперативной памяти косвенно – он содержит индекс дескриптора в таблице GTD, в котором содержится адрес таблицы LDT и ее размер.

Процесс обращается к физической памяти по виртуальному адресу, представляющему собой пару – селектор и смещение. Селектор определяет номер сегмента, а смещение – положение искомого адреса относительно начала сегмента. Селектор состоит из трех полей (рис. 6.22). Индекс задает пользовательский номер дескриптора в таблице GDT или LDT (всего 213 = 8 К сегментов). Таким образом, виртуальное адресное пространство процесса состоит из 8 К локальных и 8 К глобальных сегментов, всего из 16 К сегментов. Учитывая, что каждый сегмент имеет максимальный размер 4 Гбайт при чисто сегментной частосегментной? организации виртуальной памяти (без включения страничного механизма), процесс может работать в виртуальном адресном пространстве в 64 Тбайт.



**Рис. 6.22.**  Сегментно-страничная организация памяти в Windows

Поле из двух битов селектора задает требуемый уровень привилегий, и используется механизм защиты. В системах на базе микропроцесса Pentium поддерживается 4 уровня защиты, где уровень 0 является наиболее привилегированным, а уровень 3 – наименее привилегированным. Эти уровни образуют так называемые кольца защиты (рис. 6.23).



**Рис. 6.23.**  Кольца защиты в Windows

Система защиты манипулирует несколькими переменными, характеризующими уровень привилегий:

* DPL (Descriptor Privilege Level) – задается полем DPL в дескрипторе сегмента;
* RPL (Requested Privilege Level) – запрашиваемый уровень привилегий, задается полем RPL селектора сегмента;
* CPL (Current Privilege Level) – текущий уровень привилегий выполняемого кода, задается полем RPL селектора кодового сегмента;
* EPL (Effective Privilege Level) – эффективный уровень привилегий запроса.

Под запросом понимается любое обращение к памяти. Уровни привилегий DPL и RPL назначаются операционной системой при создании новых процессов и во время их загрузки в память. Уровень привилегий определяет не только возможности доступа к сегментам и дескрипторам, но и разрешенный набор инструкций. В каждый момент времени работающая программа находится на определенном уровне, что отмечается 2-битовым полем в регистре слова состояние программы (PSW). Уровень привилегий кодового сегмента DPL определяет текущий уровень привилегий CPL, фиксируемый в PSW.

Контроль доступа процесса к сегментным данным осуществляется на основе сопоставления эффективного уровня привилегий EPL запроса и уровня привилегий DPL дескриптора сегмента данных. Доступ может быть разрешен, если:

EPL <= DPL,

где EPL = max {CPL, RPL}.

Значение RPL – уровня запрашиваемых привилегий – определяется полем RPL -селектора, указывающего на запрашиваемый сегмент.