Московский Государственный Технический Университет имени Н. Э. Баумана

Пугачев Е.К.

Исследование методов организации оперативной памяти.

Методические указания по выполнению лабораторной работы

по дисциплине "Операционные системы".

Москва 2013

Управление оперативной памятью является важной задачей, от решения которой зависят в целом возможности ОС, например, эффективность работы многих процессов и др.

<u>Цель работы</u> - исследование способов структуризации и методов управления оперативной памяти.

<u>Продолжительность работы</u> - 5 часов.

<u>Краткие теоретические сведения</u>

Основной ресурс системы, распределением которого занимается ОС - это оперативная память. Поэтому организация памяти оказывает большое влияние на структуру и возможности ОС. В настоящее время сложилась даже более интересная ситуация - переносимая операционная система UNIX, рассчитанная на машины со страничным диспетчером памяти, произвела жесткий отбор, и теперь практически все серьезные машины, начиная от i386 и заканчивая суперкомпьютерами или, скажем процессором Alpha, имеют именно такую организацию памяти.

Самый простой случай управления памятью - ситуация, когда диспетчер памяти отсутствует, и в системе может быть загружена только одна программа. Именно в таком режиме работают CP/M и RT-11 SJ (Single-Job, однозадачная).

В этих системах программы загружаются с фиксированного адреса PROG_START. В CP/M это 0x100; в RT-11 - 01000. В адресах от 0 до начала программы находятся вектора прерываний, а в RT-11 - также и стек программы. Сама система размещается в старших адресах памяти. Адрес SYS_START, с которого она начинается, зависит от количества памяти у машины и от конфигурации самой ОС.

В этом случае управление памятью со стороны системы состоит в том, что загрузчик проверяет, поместится ли загружаемый модуль в пространство от PROG_START до SYS_START. Если объем памяти, который использует программа, не будет меняться во время ее исполнения, то на этом все управление и заканчивается.

Однако программа может использовать динамическое управление памятью, например функцию malloc() или что-то в этом роде. В этом случае уже код malloc() должен следить за тем, чтобы не залезть в системные адреса. Как правило, динамическая память начинает размещаться с адреса PROG_END = PROG_START + PROG_SIZE. PROG_SIZE в данном случае обозначает полный размер программы, то есть размер ее кода, статических данных и области, выделенной под стек.

Функция malloc() поддерживает некоторую структуру данных, следящую за тем, какие блоки памяти из уже выделенных были освобождены. При каждом новом запросе она сначала ищет блок подходящего размера в своей структуре данных и, только когда этот поиск завершится неудачей, откусывает новый блок памяти у системы. Для этого используется переменная, которая в библиотеке языка C называется brklevel. Изначально эта переменная равна PROG_END, ее значение увеличивается при выделении новых блоков, но в некоторых случаях может и уменьшаться. Это происходит, когда программа освобождает блок, который заканчивается на текущем значении brklevel.

Динамическое распределение памяти (его еще иногда называют управлением кучей (pool или heap)) представляет собой нетривиальную проблему. Действительно, активное использование функций malloc/free может привести к тому, что вся доступная память будет разбита на блоки маленького размера, и попытка выделения большого блока завершится неудачей, даже если сумма длин маленьких блоков намного больше требуемой. Это явление называется фрагментацией памяти. Кроме того, большое количество блоков требует длительного поиска. Существует также много мелких трудностей разного рода. К счастью, человечество занимается проблемой распределения памяти уже давно, и найдено много хороших или приемлемых решений.

В зависимости от решаемой задачи используются различные алгоритмы поиска свободных блоков памяти. Действительно, программа может требовать множество блоков одинакового размера, или нескольких фиксированных размеров. Это сильно облегчает решение проблемы фрагментации и поиска. Возможны ситуации, когда блоки освобождаются в порядке, обратном тому, в котором они выделялись. Это позволяет свести выделение памяти к стековой структуре. Возможны ситуации, когда некоторые из занятых блоков можно переместить по памяти. Так, например, функцию realloc() в ранних реализациях системы UNIX можно было использовать именно для этой цели.

В стандартных библиотеках языков высокого уровня, таких как malloc/free/realloc в C, new/dispose в Pascal и т.д., как правило, используются алгоритмы, рассчитанные на худший случай: программа требует блоки случайного размера в случайном порядке и освобождает их также случайным образом. Возможен, правда, более неприятный случай:

Example:

```
while(TRUE) {
    void * b1 = malloc(random(10));
    /* Случайный размер от 0 до 10 байт */
    void * b2 = malloc(random(10)+10);
    /*   ........    от 10 до 20 байт */
```

```
    if(b1 == NULL && b2 == NULL) /* Если памяти нет */
      break;                  /* выйти из цикла  */

    free(b1);
  }
  void * b3 = malloc(150);

  /* Скорее всего, память не будет выделена */
```

В результате исполнения такой программы вся доступная память будет ``порезана на лапшу'': между любыми двумя свободными блоками будет размещен занятый блок меньшего размера. К счастью, пример носит искусственный характер. В обычных программах такая ситуация встречается редко, и часто оказывается проще исправить программу, чем вносить изменения в универсальный алгоритм управления кучей.

Варианты алгоритмов распределения памяти исследовались еще в 50-е годы. Итоги многолетнего изучения этой проблемы приведены в "Кнут. Искусство программирования" и многих других учебниках.

Возможны алгоритмы распределения памяти двух типов: когда размер блока является характеристикой самого блока, и когда его сообщают отдельно при освобождении. К первому типу относится malloc/free, ко второму - GetMem/FreeMem в Turbo Pascal. В первом случае с каждым блоком ассоциируется некоторый дескриптор, который содержит длину этого блока и, возможно, какую-то еще информацию. Этот дескриптор может храниться отдельно от блока, или быть его заголовком. Иногда такой дескриптор ``окружает'' блок, то есть состоит из двух меток - в начале блока и в его конце. Для чего это может быть полезно, будет обсуждаться ниже.

Обычно все свободные блоки памяти объединяются в двунаправленный связанный список. Список должен быть двунаправленным для того, чтобы из него в любой момент можно было извлечь любой блок. Впрочем, если все действия по извлечению блока производятся после поиска, то можно слегка усложнить процедуру поиска и всегда сохранять указатель на предыдущий блок. Это решает проблему извлечения и можно ограничиться однонаправленным списком. Беда только в том, что многие алгоритмы при объединении свободных блоков извлекают их из списка в соответствии с адресом, поэтому для таких алгоритмов двунаправленный список остро необходим.

При первом взгляде на проблему возникает желание отсортировать список по размеру блока. На самом деле это бессмысленно: время поиска в сортированном списке улучшается всего в два раза по сравнению с несортированным (вот в массиве или в дереве - совсем другое дело), зато добавляется время вставки в список, пропорциональное O(n),

где n - размер списка. Помещать блоки в сортированный массив еще хуже - время вставки

становится и появляется ограничение на количество блоков. Использование хэш-таблиц или двоичных деревьев требует больших накладных расходов и усложнений программы, которые себя в итоге не оправдывают. Поэтому используют несортированный список.

Поиск в списке может вестись двумя способами: до нахождения первого подходящего (first fit) блока или до блока, размер которого ближе всего к заданному - наиболее подходящего (best fit). Для нахождения наиболее подходящего мы обязаны просматривать весь список, в то время как первый подходящий может оказаться в любом месте, и среднее время поиска будет меньше. Насколько меньше - зависит от отношения количества подходящих блоков к общему количеству. (Читатели, знакомые с теорией вероятности, могут самостоятельно вычислить эту зависимость).

Кроме того, в общем случае best fit увеличивает фрагментацию памяти. Действительно, если мы нашли блок с размером больше заданного, мы должны отделить ``хвост'' и пометить его как новый свободный блок. Понятно, что в случае best fit средний размер этого хвоста будет маленьким, и мы в итоге получим большое количество мелких блоков, которые невозможно объединить, так как пространство между ними занято.

При использовании first fit с линейным двунаправленным списком возникает специфическая проблема. Если каждый раз просматривать список с одного и того же места, то большие блоки, расположенные ближе к началу, будут чаще удаляться. Соответственно, мелкие блоки будут иметь тенденцию скапливаться в начале списка, что увеличит среднее время поиска. Простой способ борьбы с этим явлением состоит в том, чтобы просматривать список то в одном направлении, то в другом. Более радикальный и еще более простой метод состоит в том, что список делается кольцевым, и поиск каждый начинается с того места, где мы остановились в прошлый раз. В это же место добавляются освободившиеся блоки. В результате список очень эффективно перемешивается и никакой ``антисортировки'' не возникает.

В ситуациях, когда мы размещаем блоки нескольких фиксированных размеров, алгоритмы best fit оказываются лучше. Однако библиотеки распределения памяти рассчитывают на худший случай, и в них обычно используются алгоритмы first fit.

В случае работы с блоками нескольких фиксированных размеров напрашивается такое решение: создать для каждого типоразмера свой список. Это избавляет программиста от необходимости выбирать между first и best fit, устраняет поиск в списках как явление... короче, решает сразу много проблем.

Интересный вариант этого подхода для случая, когда различные размеры являются степенями числа 2, как 512 байт, 1Кбайт, 2Кбайта и т.д., называется алгоритмом близнецов. Он состоит в том, что мы ищем блок требуемого размера в соответствующем списке. Если этот список пуст, мы берем список блоков вдвое большего размера. Получив блок вдвое большего размера, мы делим его пополам. Ненужную половину мы помещаем в соответствующий список свободных блоков. Любопытно, что нам совершенно неважно, получили ли мы этот блок просто из соответствующего списка, или же делением пополам вчетверо большего блока, и далее по рекурсии. Одно из преимуществ этого метода состоит в простоте объединения блоков при их освобождении. Действительно, адрес блока-близнеца получается простым инвертированием соответствующего бита в адресе нашего блока. Нужно только проверить, свободен ли этот близнец. Если он свободен, то мы объединяем братьев в блок вдвое большего размера, и т.д.

Алгоритм близнецов значительно снижает фрагментацию памяти и резко ускоряет поиск блоков. Наиболее важным преимуществом этого подхода является то, что даже в наихудшем случае время поиска не превышает , где и обозначают соответственно максимальный и минимальный размеры используемых блоков. Это делает алгоритм близнецов трудрозаменимым для ситуаций, когда необходимо гарантированное время реакции - например, для задач реального времени. Часто этот алгоритм или его варианты используются для выделения памяти внутри ядра ОС. Например, функция kmalloc, используемая в ядре ОС Linux, основана именно на алгоритме близнецов.

Разработчик программы динамического распределения памяти обязан решить еще одну важную проблему, а именно - объединение свободных блоков. Действительно, обидно, если мы имеем сто свободных блоков по одному килобайту и не можем сделать из них один блок в сто килобайт. Но если все эти блоки расположены в памяти один за другим, а мы не можем их при этом объединить - это просто унизительно. Кроме того, если мы умеем объединять блоки и видим, что объединенный блок ограничен сверху значением brklevel, то мы можем вместо помещения этого блока в список просто уменьшить значение brklevel и, таким образом, вернуть ненужную память системе.

Рассмотрим способы решения этой проблемы. Один способ упоминался в описании алгоритма близнецов, но он пригоден только в особых случаях.

Представим себе для начала, что все, что мы знаем о блоке, - это его начальный адрес и размер. Такая ситуация возможна при выделении памяти ``второго рода'', т.е. когда размер освобождаемого блока передается как параметр процедуры FreeMem. Кроме того, такое возможно и в случаях, когда дескриптор блока содержит только его длину.

Легко понять, что это очень плохая ситуация. Действительно, для объединения блока с соседями мы должны найти их в списке свободных, или же убедиться, что там их нет. Для этого мы должны просмотреть весь список. В порядке мозгового штурма можно высказать идею сортировать список свободных блоков по адресу...\
Гораздо проще запоминать в дескрипторе блока указатели на дескрипторы соседних блоков. Немного развив эту идею, мы приходим к методу, который упоминался в начале раздела. Этот метод называется алгоритмом парных меток и состоит в том, что мы добавляем к каждому блоку по два слова памяти.

*Именно слова, а не байта. Дело в том, что требуется добавить достаточно места, чтобы хранить там размер блока в байтах или словах. Обычно такое число занимает столько же места, сколько и адрес, а размер слова обычно равен размеру адреса. На x86 в реальном режиме это не так, но это вообще довольно странный процессор.
*
Итак, мы добавляем к блоку два слова - одно спереди, другое сзади. В оба слова мы записываем размер блока. Это и есть тот дескриптор, который окружает блок. При этом мы говорим, что значения длины будут положительными, если блок свободен, и отрицательными, если блок занят. Можно сказать и наоборот, важно только потом соблюдать это соглашение.
Представим, что мы освобождаем блок с адресом addr. Считаем, что addr имеет тип word *, и при добавлении к нему целых чисел результирующий адрес будет отсчитываться в словах, как в языке C. Для того чтобы проверить, свободен ли его сосед спереди, мы должны посмотреть слово с адресом addr - 2. Если оно отрицательно, то сосед занят, и мы должны оставить его в покое. Если же оно положительно, то мы можем легко определить адрес начала этого блока как addr - addr[-2]. Определив адрес начала блока, мы можем легко объединить этот блок с блоком addr, нам нужно только сложить значения меток-дескрипторов и записать их в дескрипторы нового большого блока. Нам даже не нужно будет добавлять освобождаемый блок в список и извлекать оттуда его соседа!
Похожим образом присоединяется и сосед сзади. Единственное отличие состоит в том, что этого соседа все-таки нужно извлекать из списка свободных блоков.
Дополнительное преимущество приведенного алгоритма состоит в том, что мы можем отлавливать такие ошибки, как многократное освобождение одного блока, запись в память за границей блока и иногда даже обращение к уже освобожденному блоку. Действительно, мы в любой момент можем проверить всю цепочку блоков памяти и убедиться в том, что все свободные блоки стоят в списке, что в нем стоят только свободные блоки, что сами цепочка и список не испорчены, и т.д.

* Это действительно большое преимущество, так как оно значительно облегчает ловлю ошибок работы с указателями, о которых в руководстве по Zortech C/C++ сказано, что ``опытные программисты, услышав это слово [``pointer bug'' - прим. авт.], бледнеют и прячутся под стол''.

*

Итак, наилучшим из известных универсальных алгоритмов динамического распределения памяти является алгоритм парных меток с объединением свободных блоков в двунаправленный кольцевой список и поиском по принципу first fit. Этот алгоритм обеспечивает приемлемую производительность почти для всех стратегий распределения памяти, используемых в прикладных программах. Такой алгоритм используется практически во всех реализациях стандартной библиотеки языка С и во многих других ситуациях. Другие известные алгоритмы либо просто хуже, чем этот, либо проявляют свои преимущества только в специальных случаях.

К основным недостаткам этого алгоритма относится отсутствие верхней границы времени поиска подходящего блока, что делает его неприемлемым для задач реального времени.

Некоторые системы программирования используют специальный метод освобождения динамической памяти, называемый сборкой мусора. Этот метод состоит в том, что ненужные блоки памяти не освобождаются явным образом. Вместо этого используется некоторый более или менее изощренный алгоритм, следящий за тем, какие блоки еще нужны, а какие - уже нет.

Самый простой метод отличать используемые блоки от ненужных - считать, что блок, на который есть ссылка, нужен, а блок, на который ни одной ссылки не осталось - не нужен. Для этого к каждому блоку присоединяют дескриптор, в котором подсчитывают количество ссылок на него. Каждая передача указателя на этот блок приводит к увеличению счетчика ссылок на 1, а каждое уничтожение объекта, содержавшего указатель - к уменьшению.

Впрочем, при таком подходе возникает специфическая проблема. Если у нас есть циклический список, на который нет ни одной ссылки извне, то все объекты в нем будут считаться используемыми, хотя они и являются мусором. Если мы по тем или иным причинам уверены, что кольца не возникают или возникают очень редко, метод подсчета ссылок вполне приемлем; если же мы используем графы произвольного вида, необходим более умный алгоритм.

Все остальные методы сборки мусора так или иначе сводятся к поддержанию базы данных о том, какие объекты на кого ссылаются. Использование такой техники возможно

практически только в интерпретируемых языках типа Lisp или Prolog, где с каждой операцией можно ассоциировать неограниченно большое количество действий.

Описанные выше алгоритмы распределения памяти используются не операционной системой, а библиотечными функциями, пришитыми к программе. Однако многозадачная или многопрограммная ОС также должны использовать тот или иной алгоритм размещения памяти.

Отчасти такие алгоритмы могут быть похожи на работу malloc. Однако режим работы ОС может вносить существенные упрощения в алгоритм.

Так, например, процедура управления памятью MS DOS рассчитана на случай, когда программы выгружаются из памяти только в порядке, обратном тому, в каком они туда загружались. Это позволяет свести управление памятью к стековой дисциплине.

Каждой программе в MS DOS отводится блок памяти. С каждым таким блоком ассоциирован дескриптор, называемый MCB - Memory Control Block. Этот дескриптор содержит размер блока, идентификатор программы, которой принадлежит этот блок и признак того, является ли данный блок последним в цепочке. Нужно отметить, что программе всегда принадлежит несколько блоков, но это уже несущественные детали. Другая малосущественная деталь та, что размер сегментов и их адреса отсчитываются в параграфах размером 16 байт. Знакомые с архитектурой процессоров 80x86 должны понять, что адрес MCB в этом случае будет состоять только из сегментной части с нулевым смещением.

* Идентификатор программы в MS DOS может служить темой для отдельного разговора. Во многих документах он носит громкое название pid (идентификатор процесса) - почти как в UNIX, но на самом деле это всего лишь адрес PSP (Program Segment Prefix).

*

После запуска .com-файл получает сегмент размером 64К, а .exe - всю доступную память. Обычно .exe-модули сразу после запуска освобождают ненужную им память и устанавливают brklevel на конец своего сегмента, а потом увеличивают brklevel и наращивают сегмент по мере необходимости. Естественно, что наращивать сегмент можно только за счет следующего за ним в цепочке MCB, и MS DOS разрешит делать это только в случае, если этот сегмент не принадлежит никакой программе.

При запуске программы DOS берет последний сегмент в цепочке, и загружает туда программу, если этот сегмент достаточно велик. Если он недостаточно велик, DOS говорит Not enough memory и отказывается загружать программу.

При завершении программы DOS освобождает все блоки, принадлежавшие программе. При этом соседние блоки объединяются. Пока программы, действительно, завершаются в

порядке, обратном тому, в котором они запускались, - все вполне нормально. Другое дело, что в реальной жизни возможны отклонения от этой схемы.

Например, неявно предполагается, что TSR-программы (Terminate, but Stay Resident) никогда не пытаются завершиться. Тем не менее любой уважающий себя хакер считает своим долгом сделать резидентную программу выгружаемой. У некоторых хакеров она, в результате, выбрасывает при выгрузке все резиденты, которые сели в память после них. Другой пример - отладчики обычно загружают программу в обход обычной DOS-овской функции LOAD & EXECUTE, а при завершении отлаживаемой программы сами освобождают память из-под нее.

Один из авторов в свое время занимался прохождением некоторой программы под отладчиком. Честно говоря, речь шла о взломе некоторой игрушки... Эта программа производила какую-то инициализацию, а потом вызывала функцию DOS LOAD & EXECUTE. Я об этом не знал и, естественно, провалился внутрь новой программы, которую и должен был, по-хорошему, взламывать. После нескольких нажатий <CTRL> <Break> я наконец-то попал в отладчик, но при каком-то очень странном состоянии программы. Полазив по программе некоторое время и убедившись, что она не хочет приходить в нормальное состояние, я вышел из отладчика и увидел следующую картину: системе доступно около 100Кбайт в то время, как сумма длин свободных блоков памяти более 300Кбайт, а размер наибольшего свободного блока около 200Кбайт. Отладчик, выходя, освободил свою память и память отлаживаемой программы, но не освободил память из-под нового загруженного модуля. В результате посредине памяти остался никому не нужный блок памяти изрядного размера, помеченный как используемый. Самым обидным было то, что DOS не пыталась загрузить ни одну программу в память под этим блоком, хотя там было гораздо больше места, чем над ним.

В качестве итога можно сказать, что основными проблемами многопрограммных систем без диспетчера памяти являются:

- Проблема выделения дополнительной памяти программе, которая загружалась не последней.
- Проблема освобождения памяти после завершения программы. В системах с монолитным загружаемым модулем иногда просто запрещают программам выгружаться. В MS DOS была сделана попытка запретить выгружать TSR и драйверы, но это привело к поиску задних дверей.
- Низкая надежность. Ошибка в одной из программ может привести к порче кода или данных других программ или самой системы.
- Проблемы безопасности. В системах с открытой памятью невозможны эффективные средства разделения доступа. Любая программная проверка прав доступа может быть

легко обойдена прямым вызовом ``защищаемых" модулей ядра. Даже криптографические средства не обеспечивают достаточно эффективной защиты, потому что можно посадить в память троянскую программу, которая будет анализировать код программы шифрования и считывать значение ключа...

В системах с динамической сборкой первые две проблемы не так остры, потому что память выделяется и освобождается небольшими кусочками, по блоку на каждый объектный модуль, поэтому код программы обычно не занимает непрерывного пространства. Соответственно, такие системы часто разрешают и данным программы занимать несмежные области памяти.

Такой подход используется многими системами с открытой памятью - AmigaDOS, Oberon, системами программирования для транспьютера и т.д. Однако в таких системах очень острую форму приобретает проблема фрагментации свободной памяти.

Количество фрагментов приблизительно пропорционально количеству операций выделения/освобождения памяти. Вероятность не найти блок подходящего размера из-за фрагментации оценить сложнее, но она также растет с количеством операций.

Понятно, что если эти операции осуществляются сразу многими программами, то фрагментация пропорционально возрастает. Вышеперечисленные системы не предоставляют никаких средств для борьбы с фрагментацией.

Напротив, в системе MacOS было предложено достаточно оригинальный метод борьбы с фрагментацией. Метод этот заслуживает отдельного обсуждения.

В этих системах предполагается, что пользовательские программы не сохраняют указателей на динамически выделенные блоки памяти. Вместо этого каждый такой блок идентифицируется целочисленным дескриптором или ``ручкой" (handle). Когда программа непосредственно обращается к данным в блоке, она выполняет системный вызов GlobalLock (запереть). Этот вызов возвращает текущий адрес блока. Пока программа не исполнит вызов GlobalUnlock (отпереть), система не пытается изменить адрес блока. Напротив, если блок не заперт, система считает себя вправе передвигать его по памяти или даже сбрасывать на диск.

Таким образом, ``ручки" представляют собой попытку создать программный аналог аппаратных диспетчеров памяти. Они позволяют решить проблему фрагментации и даже организовать некое подобие виртуальной памяти. Можно рассматривать их как средство организации оверлейных данных - поочередного отображения разных блоков данных на одни и те же адреса. Однако за это приходится платить очень дорогой ценой.

Использование ``ручек" сильно усложняет программирование вообще и в особенности перенос ПО из систем, использующих линейное адресное пространство. Все указатели на динамические структуры данных в программе нужно заменить на ``ручки", а каждое

обращение к таким структурам необходимо окружить вызовами GlobalLock/GlobalUnlock. Эти вызовы:

- сами по себе увеличивают объем кода и время исполнения;
- мешают компиляторам выполнять оптимизацию, прежде всего не позволяют оптимально использовать регистры процессора, потому что далеко не все регистры сохраняются при вызовах;
- требуют разрыва конвейера команд и перезагрузки командного кэша; в современных суперскалярных процессорах это может приводить к падению производительности во много раз.

Попытки уменьшить число блокировок требуют определенных интеллектуальных усилий. Фактически, к обычному циклу разработки ПО: проектирование, выбор алгоритма, написание кода и его отладка - добавляется еще две фазы: микрооптимизация использования ``ручек'' и отладка оптимизированного кода. Последняя фаза оказывается, пожалуй, самой сложной и ответственной.

Наиболее неприятной и опасной ошибкой, возникающей на фазе микрооптимизации, является вынос указателя на динамическую структуру за пределы скобок GlobalLock/GlobalUnlock. Эту ошибку очень сложно обнаружить при тестировании, так как она проявляется только если система пыталась передвигать блоки в промежутках между обращениями. Иными словами, ошибка может проявлять или не проявлять себя в зависимости от набора приложений, исполняющихся в системе и от характера деятельности этих приложений. В результате мы получаем то, чего больше всего боятся инженеры, механики и программисты - систему, которая работает иногда.

Не случайно фирма MicroSoft полностью отказалась от ``ручечного'' управления памятью в новой версии MS Windows - Windows 95, где реализована полноценная виртуальная память.

Как уже говорилось, в системах с открытой памятью возникают большие сложности при организации многозадачной работы. Чтобы устранить их, необходимо предоставлять каждой задаче свое виртуальное адресное пространство. Наиболее простым способом организовать различные адресные пространства является так называемая базовая адресация. По-видимому, это исторически наиболее ранний способ.

Вы можете заметить, что термин базовая адресация уже занят - мы называли таким образом адресацию по схеме reg[offset]. Дело в том, что метод, о котором сейчас идет речь, состоит в формировании адреса по той же схеме. Отличие состоит в том, что регистр, относительно которого происходит адресация, не доступен прикладной программе. Кроме того, его значение прибавляется ко всем адресам, в том числе к ``абсолютным'' адресным ссылкам или переменным типа указатель. По существу, такая адресация является

способом организации виртуального адресного пространства. Действительно, распечатав значение указателя, вы увидите число, равное, например, 0x000A25. Однако такой адрес будет соответствовать физическому адресу BASE + 0x000A25 и меняться вместе со значением регистра BASE.

Если говорить о разнице между двумя значениями слов ``базовая адресация'', нельзя не упомянуть о поучительной истории, связанной с управлением памятью в системах линии IBM System 360.

В этих машинах не было аппаратных средств управления памятью и все программы разделяли общее виртуальное адресное пространство, совпадающее с физическим. Адресные ссылки в программе задавались 12-битовым смещением относительно базового регистра. В качестве базового регистра мог использоваться, в принципе, любой из 16 32-битных регистров общего назначения. Предполагалось, что пользовательские программы не модифицируют базовый регистр, поэтому можно загружать их с различных адресов просто перенастраивая значение этого регистра. Таким образом была реализована одновременная загрузка многих программ в многозадачной системе OS/360.

Однако после загрузки программу уже нельзя было перемещать по памяти: например при вызове подпрограммы адрес возврата сохраняется в стеке в виде абсолютного 24-битного адреса и при возврате базовый регистр не используется. Аналогично, базовый регистр не используется при ссылках на параметры подпрограмм языка FORTRAN, при работе с указателями в PL/I и т.д.. Перемещение программы, даже с перенастройкой базового регистра, нарушило бы все такие ссылки.

Разработчики фирмы IBM вскоре осознали пользу перемещения программ после их загрузки и попытались как-то решить эту проблему. Очень любопытный документ 'Preparing to Rollin-Rollout Guideline' описывает действия, которые программа должна была бы предпринять после перемещения. Фактически программа должна была найти в своем сегменте данных все абсолютные адреса и сама перенастроить их.

Естественно, никто из разработчиков компиляторов и прикладного программного обеспечения не собирался следовать этому руководству. В результате проблема перемещения программ в OS/360 не была решена вплоть до появления машин System 370 со страничным или странично-сегментным диспетчером памяти и ОС MVS.

Как правило, машины, использующие базовую адресацию, имеют два регистра. Один из регистров задает базу для адресов, второй устанавливает верхний предел. В системе ICL1900/Одренок эти регистры называются соответственно BASE и DATUM. Если адрес выходит за границу, установленную значением DATUM, возникает исключительная ситуация (exception) ошибочной адресации. Как правило, это приводит к тому, что система принудительно завершает работу программы.

При помощи этих двух регистров мы сразу решаем две важные проблемы.

Во-первых, мы можем изолировать программы друг от друга - ошибки в одной программе не приводят к разрушению или повреждению других программ или самой системы. Благодаря этому мы можем обеспечить защиту системы не только от ошибочных программ, но и от злонамеренных действий пользователей по разрушению системы или доступу к чужим данным.

Во-вторых, мы получаем возможность передвигать адресные пространства задач по физической памяти так, что сама программа не замечает, что ее передвинули. За счет этого мы решаем проблему фрагментации памяти и даем программам возможность наращивать свое адресное пространство. Действительно, в системе с открытой памятью программа может добавлять себе память только до тех пор, пока не упрется в начало следующей программы. После этого мы должны либо говорить, что памяти нет, либо мириться с тем, что программа может занимать несмежные области физического адресного пространства. Второе решение резко усложняет управление памятью, как со стороны системы, так и со стороны программы, и часто оказывается неприемлемым (подробнее связанные с этим проблемы обсуждаются в разделе 3.3). В случае же базовой адресации мы можем просто сдвинуть мешающую нам программу вверх по физическим адресам.

Часто системы, работающие на таких архитектурах, умеют сбрасывать на диск те задачи, которые долго не будут исполняться. Это самая простая из форм своппинга (swapping - обмен).

В современных системах базовая виртуальная адресация используется редко. Дело не в том, что она плоха, а в том, что более сложные методы, такие как сегментная и страничная трансляция адресов, оказались намного лучше. Часто под словами ``виртуальная память'' подразумевают именно сегментную или страничную адресацию.

В системах с сегментной и страничной адресацией виртуальный адрес имеет сложную структуру. Он разбит на два битовых поля: номер страницы (сегмента) и смещение в нем. Соответственно, адресное пространство оказывается состоящим из дискретных блоков. Если все эти блоки имеют фиксированную длину и образуют вместе непрерывное пространство, они называются страницами. Если длина каждого блока может задаваться произвольно, а неиспользуемым частям блоков соответствуют ``дыры'' в виртуальном адресном пространстве, они называются сегментами. Как правило, один сегмент соответствует коду или данным одного модуля программы. Со страницей или сегментом могут быть ассоциированы права чтения, записи и исполнения.

Такая адресация реализуется аппаратно. Процессор, как правило, имеет специальное устройство, называемое диспетчером памяти. В некоторых процессорах, например в MC68020 или MC68030 или в некоторых RISC-системах, это устройство реализовано на

отдельном кристалле; в других, таких как i386 или Alpha, диспетчер памяти интегрирован в процессор.

Диспетчер памяти содержит регистр - указатель на таблицу трансляции. Эта таблица размещается где-то в физической памяти. Ее элементами являются дескрипторы каждой страницы/сегмента. Такой дескриптор содержит права доступа к странице, признак присутствия этой страницы в памяти и физический адрес страницы/сегмента. Для сегментов в дескрипторе также хранится длина сегмента.

Как правило, диспетчер памяти имеет также кэш (cache) дескрипторов - быструю память с ассоциативным доступом. В этой памяти хранятся дескрипторы часто используемых страниц.

Алгоритм доступа к памяти по виртуальному адресу page:offset получается следующим:

- Проверить, существует ли страница page вообще. Например, у машин семейства VAX адресное пространство может состоять из 8M страниц, что соответствует 4 Гбайтам, но реальные программы используют намного меньше памяти, и размеры их таблиц трансляции соответственно уменьшаются. Естественно, про страницу, для которой нет соответствующего элемента таблицы, можно сказать, что ее не существует. Такая проверка осуществляется простым сравнением номера страницы с длиной таблицы трансляции. Если страницы не существует, возникает особая ситуация ошибки сегментации (segmentation violation)
- Попытаться найти дескриптор страницы в кэше.
- Если его нет в кэше, загрузить дескриптор из таблицы в памяти.
- Проверить, имеет ли процесс соответствующее право доступа к странице. Иначе также возникает ошибка сегментации.
- Проверить, находится ли страница в оперативной памяти. Если ее там нет, возникает особая ситуация отсутствия страницы или страничный отказ (page fault). Как правило, реакция на нее состоит в том, что вызывается специальная программа-обработчик (trap - ловушка), которая подкачивает требуемую страницу с диска. В многозадачных системах во время такой подкачки может исполняться другой процесс.
- Если страница есть в памяти, взять из ее дескриптора физический адрес phys_addr.
- Если мы имеем дело с сегментной адресацией, сравнить смещение в сегменте с длиной этого сегмента. Если смещение оказалось больше, также возникает ошибка сегментации.
- Произвести доступ к памяти по адресу phys_addr[offset].

Видно, что такая схема адресации довольно сложна. Однако в современных процессорах все это реализовано в аппаратуре и, благодаря кэшу дескрипторов и другим ухищрениям, скорость доступа к памяти получается почти такой же, как и при прямой адресации. Кроме того, эта схема имеет неоценимые преимущества при реализации многозадачных ОС.

Во-первых, мы можем связать с каждой задачей свою таблицу трансляции, а значит и свое виртуальное адресное пространство. Благодаря этому даже в многозадачных ОС мы можем пользоваться абсолютным загрузчиком. Кроме того, программы оказываются изолированными друг от друга, и мы можем обеспечить их безопасность.
Для переключения задачи нужно перегрузить регистры, управляющие этой таблицей.

* Это не один регистр, а, как минимум, два - указатель на таблицу и ее длина. В большинстве современных процессоров диспетчер памяти еще сложнее. Часто он работает с несколькими таблицами, может иметь дополнительные управляющие регистры и т.д.
*
Во-вторых, мы можем сбрасывать на диск редко используемые области виртуальной памяти программ - не всю программу целиком, а только ее часть. Кроме того, в отличие от оверлеев, программа вообще не обязана знать, какая ее часть может быть сброшена.

* Другое дело, что в системах реального времени программе может быть нужно, чтобы определенные ее части никогда не сбрасывались на диск. Действительно, система реального времени обязана гарантировать время реакции, и это гарантированное время обычно намного меньше времени доступа к диску. Естественно, что код, обрабатывающий событие, и используемые при этом данные должны быть всегда в памяти.
*
В-третьих, программа не обязана занимать непрерывную область физической памяти. При этом она вполне может видеть непрерывное виртуальное адресное пространство. Это резко упрощает борьбу с фрагментацией памяти, а в системах со страничной адресацией проблема фрагментации физической памяти вообще снимается.

* Так, в VAX/VMS свободная память отслеживается при помощи битовой маски физических страниц. В этой маске свободной странице соответствует 1, а занятой - 0. Если кому-то нужна страница, система просто ищет в этой маске установленный бит. Если учесть, что VAX имеет специальную команду для поиска установленного бита в массиве, все работает очень быстро и просто.
В результате виртуальное пространство программы может оказаться отображено на физические адреса очень причудливым образом, но это никого не волнует - скорость доступа ко всем страницам одинакова.
*
В-четвертых, система может обеспечивать не только защиту программ друг от друга, но и защиту программы от самой себя - например, от ошибочной записи данных на место кода.

В-пятых, различные задачи могут использовать общие области памяти для взаимодействия или, скажем, просто для того, чтобы работать с одной копией библиотеки подпрограмм.

Перечисленные преимущества настолько серьезны, что считается невозможным реализовать многозадачную систему общего назначения, такую как UNIX или VMS (Virtual Memory System) на машинах без диспетчера памяти.

Отдельной проблемой при разработке системы со страничной или сегментной адресацией является выбор размера страницы или максимального размера сегмента. Этот размер определяется шириной соответствующего битового поля адреса и поэтому должен быть степенью двойки.

С одной стороны, страницы не должны быть слишком большими, так как это может привести к неэффективному использованию памяти и перекачке слишком больших объемов данных при сбросе страниц на диск. С другой стороны, страницы не должны быть слишком маленькими, так как это приведет к чрезмерному увеличению таблиц трансляции, требуемого объема кэша дескрипторов и т.д.

Что будет считаться ``слишком'' большим или, наоборот, маленьким, в действительности зависит от среднего количества памяти, используемого программой. Так, если вы работаете со старой UNIX-системой и пользуетесь только программами типа ls, grep или sed, а редактор vi считается чем-то почти сверхъестественным, то средний размер ваших программ будет измеряться десятками килобайт и редко выходить за сотню. Если же вы используете современные графические оконные интерфейсы, систему X Windows, emacs и прочие красоты, то типичная программа требует около мегабайта памяти, а некоторые и по несколько десятков. Казалось бы, размер страницы при этом также должен измениться на два-три порядка. На самом деле, здесь в игру вступает еще один параметр - размер сектора на диске, с которого осуществляется подкачка...\

В реальных системах размер страницы меняется от 512 байт у машин семейства VAX до нескольких килобайт. Например, i3/486 имеет страницу размером 4 К. Некоторые диспетчеры памяти, например у MC6801/2/30, имеют переменный размер страницы - в том смысле, что система при запуске программирует диспетчер и устанавливает, помимо прочего, этот размер, и дальше работает со страницами выбранного размера. У процессора i860 размер страницы переключается между 4 К и 4 Мбайтами.

С сегментными диспетчерами памяти ситуация сложнее. С одной стороны, хочется, чтобы один программный модуль влезал в сегмент, поэтому сегменты обычно делают большими, от 32К и более. С другой стороны, хочется, чтобы в адресном пространстве можно было сделать много сегментов. Кроме того, может возникнуть проблема: как быть с большими неразделимыми объектами, например хэш-таблицами компиляторов, под которые часто

выделяются сотни килобайт. С третьей стороны, при подкачке сегментов с диска не хочется качать за один раз много данных.

Третье обстоятельство вынуждает многих разработчиков идти на двухступенчатую виртуальную память - сегментную адресацию, в которой каждый сегмент, в свою очередь, разбит на страницы. Это дает ряд мелких преимуществ, например, позволяет раздавать права доступа сегментам, а подкачку с диска осуществлять постранично. Таким образом организована виртуальная память в IBM System 370 и ряде других больших компьютеров, а также в i3/486. Правда, в последнем виртуальная память используется несколько странным образом.

Страничный обмен

Вообще говоря, i386 может работать с двумя типами адресов:

- 32-разрядным адресом, в котором 16 бит задают смещение в сегменте, 14 бит - номер сегмента, и 2 бита используются для разных загадочных целей. При этом размер сегмента не более 64К, а общий объем виртуальной памяти не превышает 1 Гбайта.
- 48-разрядным адресом, в котором смещение в сегменте занимает 32 бита. В этом случае размер сегмента может быть до 4 Гбайт, а общий объем виртуальной памяти до байт.

В обоих случаях сегмент может быть разбит на страницы по 4 К. При этом сегментная часть адреса и его смещение лежат в разных регистрах, и с ними можно работать раздельно. В реальном режиме возможность такой работы порождает весь зоопарк ``моделей памяти'', с которыми знакомы те, кто писал на C для MS DOS. В случае i386 большинство систем программирования выделяют программе один сегмент с 32-разрядным смещением, и программа живет там так, будто это обычная машина с 32-разрядным линейным адресным пространством. Так поступают все известные авторам реализации Unix для i386, ряд так называемых расширителей ДОС (DOS extenders), Oberon/386, Novell Netware и т.д.

С другой стороны, MS Windows в enhanced режиме используют i386 именно как машину с двухслойной сегментно-страничной адресацией, то есть загружают программу по сегментам, и права доступа ей выдают на сегменты, а подкачку с диска делают постранично. Это обусловлено не столько настоящей потребностью в сегментации, сколько требованием совместимости со standard режимом, в котором MS Windows работают на процессоре 80286

Подкачка, или своппинг (swapping - обмен), - это процесс сброса редко используемых областей виртуального адресного пространства программы на диск или другое устройство массовой памяти. При разработке системы всегда есть желание сделать память как можно быстрее. С другой стороны, потребности в памяти очень велики и постоянно растут. Современные персональные системы имеют около 500 Мбайт дисковой памяти, и этого

часто оказывается недостаточно, особенно если идет работа с Multimedia или просто с высококачественными изображениями.

Существует эмпирическое наблюдение, что любой объем дисковой памяти будет полностью занят за две недели. Жизненный опыт обоих авторов подтверждает его.

Очевидно, что система с 500 М статического ОЗУ будет иметь стоимость, скажем так, совершенно не персональную, не говоря уже о габаритах, потребляемой мощности и прочем. К счастью, далеко не все, что хранится в памяти системы, используется одновременно. В каждый заданный момент исполняется только часть программного обеспечения, и оно работает только с частью данных.

Статистика утверждает, что в пределах одной программы 90% времени исполняется код, который занимает 10% места, а остальные 90% кода исполняются только 10% времени. Для данных разница в частоте использования, по-видимому, не столь резкая, но также существует.

Это приводит нас к идее многослойной или многоуровневой памяти, когда в быстрой памяти хранятся часто используемые код или данные, а редко используемые постепенно мигрируют на более медленные устройства. В случае дисковой памяти такая миграция осуществляется вручную, когда администратор системы сбрасывает на ленты редко используемое ``барахло'' и заполняет освободившееся место чем-то нужным. Для больших и сильно загруженных систем существуют специальные программы, которые определяют, что является барахлом, а что нет. Управление миграцией из ОЗУ на диск иногда осуществляется пользователем, но часто это оказывается слишком утомительно. В случае кэш-памяти и ОЗУ делать что-то вручную просто физически невозможно.

Естественно, для того чтобы автоматизировать процесс удаления ``барахла'' - редко используемых данных и программ, - мы должны иметь какой-то легко формализуемый критерий, по которому определяется, какие данные считаются редко используемыми. Для ручного переноса данных критерий очевиден - нужно удалять то, что дольше всего не будет использоваться в будущем. Конечно, любые предположения о будущем имеют условный характер, все может неожиданно измениться. Тем не менее, обычно человек располагает такой информацией о системе и ее использовании, которая принципиально недоступна самой этой системе. А для автомата будущее неизвестно, и он должен делать догадки о будущем только на основании данных об использовании системы в прошлом.

Алгоритм автомата должен быть как можно более простым. Например, кэш-контроллер обычно делается на основе ``жесткой логики'', потому что он должен быть очень быстрым. Даже микропрограммный автомат с точки зрения скорости оказывается недопустимым, не говоря уже о сложных алгоритмах, использующих динамические структуры данных. Для алгоритма управления страничной подкачкой простота также важна - очень не хочется,

чтобы большую часть времени система занималась размышлениями о том, какую страницу можно сбросить на диск. Сравните с армейским афоризмом: 'командир должен уметь принимать быстрое решение, а если оно случайно окажется правильным, то это будет вообще замечательно'.

Один простой критерий выбора очевиден - при прочих равных условиях, в первую очередь, мы должны выбирать в качестве жертвы (victim) для удаления тот объект, который не был изменен за время жизни в быстрой памяти. Действительно, вы скорее удалите с винчестера саму игрушку (если у вас есть ее копия на дискетах), чем файлы сохранения!

Самый простой алгоритм - выкидывать случайно выбранный объект. Такой алгоритм хорош тем, что очень прост - не надо набирать никакой статистики о частоте использования и т.д. Очевидно, при этом удаленный объект совершенно необязательно будет ненужным...\

Можно также удалять то, что дольше всего находится в данном слое памяти. Это называется алгоритмом FIFO (First In - First Out - первый вошел - первый вышел). Видно, что это уже чуть сложнее случайного удаления - нужно запоминать, когда мы что загружали. Понятно также, что это лишь очень грубое приближение к тому, что нам требуется.

Наиболее честным будет удалять тот объект, к которому дольше всего не было обращений в прошлом LRU (Least Recently Used). Это требует набора статистики обо всех обращениях. Для страничного или сегментного управления памятью это также требует аппаратной поддержки - мы ведь не в состоянии программно отслеживать все обращения ко всем страницам без катастрофического падения производительности системы.

Такая поддержка на практике должна состоять в том, что диспетчер памяти поддерживает в дескрипторе каждой страницы счетчик обращений, и при каждой операции чтения или записи над этой страницей увеличивает этот счетчик на единицу. Это требует довольно больших накладных расходов - в ряде работ утверждается, что они будут недопустимо большими. Поэтому такая техника применяется только в экспериментальных установках, используемых для оценки производительности тех или иных алгоритмов. Она также применяется в некоторых контроллерах кэш-памяти и программно реализованных дисковых кэшах.

Остроумным приближением к алгоритму LRU является так называемый clock-алгоритм. Он состоит в следующем:

- Дескриптор каждой страницы содержит бит, указывающий, что к данной странице было обращение. Этот бит иногда называют clock-битом.

- При первом обращении к странице, в которой clock-бит был сброшен, диспетчер памяти устанавливает этот бит.
- Программа, занимающаяся поиском жертвы, циклически просматривает все дескрипторы страниц. Если clock-бит сброшен, данная страница объявляется жертвой, и просмотр заканчивается - до появления потребности в новой странице. Если clock-бит установлен, то программа сбрасывает его и продолжает поиск.

Название clock, по-видимому, происходит от внешнего сходства процесса циклического просмотра с движением стрелки часов. Подумав, можно убедиться, что вероятность оказаться жертвой для страницы, к которой часто происходят обращения, существенно ниже.

Практически все известные авторам диспетчеры памяти предполагают использование clock-алгоритма. Такие диспетчеры хранят в дескрипторе страницы или сегмента два бита - clock-бит и признак модификации. Признак модификации устанавливается при первой записи в страницу/сегмент, в дескрипторе которой этот признак был сброшен.

Экспериментальные исследования показывают любопытный факт: реальная производительность системы довольно слабо зависит от применяемого алгоритма поиска жертвы. Статистика исполнения реальных программ говорит о том, что каждая программа имеет некоторый набор страниц, называемый рабочим множеством, который ей в данный момент действительно нужен. Размер такого набора сильно зависит от алгоритма программы, он изменяется на различных этапах исполнения и т.д., но в большинство моментов мы можем довольно точно указать его. Если все страницы рабочего набора попадают в память, то частота ошибок отсутствия страницы резко снижается. В случае, когда памяти не хватает, программе почти на каждой команде требуется новая страница, и производительность системы катастрофически - в тысячи раз - падает. В случае машин типа IBM PC/AT386 с дисковым контроллером IDE, в которых процессор задействуется при операциях с диском, это может привести, практически, к блокировке системы. Это состояние по-английски называется overswap или thrashing - чрезмерный своппинг и является крайне нежелательным.

В системах коллективного пользования размер памяти часто выбирают так, чтобы система балансировала где-то между состоянием, когда все программы держат свое рабочее множество в ОЗУ, и оверсвопом. Точное положение точки балансировки определяется в зависимости от соотношения скорости процессора со скоростью обмена с диском и с потребностями прикладных программ. Во многих старых учебниках рекомендуется подбирать объем памяти так, чтобы канал дискового обмена был загружен на 50%.

Еще одно эмпирическое правило приводится в документации фирмы Amdahl: сбалансированная система должна иметь по мегабайту памяти на каждый MIPS (Million of

Instructions Per Second - миллион операций в секунду) производительности центрального процессора. Если система не использует память, определенную по этой формуле, есть основания считать, что процессор также работает с недогрузкой. Иными словами, это означает, что вы купили слишком мощный для ваших целей процессор и заплатили лишние деньги.

Это правило было выработано на основе опыта эксплуатации больших компьютеров четвертого поколения, в основном на задачах управления базами данных. Скорость дисковой подсистемы в этих машинах была примерно сравнима с дисковыми контроллерами современных персоналок, поэтому аналогичный критерий оценки применим и к ПК, особенно работающим под управлением систем с виртуальной памятью - OS/2, Windows NT и системами семейства Unix.

В соответствии с этим правилом, машины с процессором i486 должны иметь 8-16 мегабайт памяти, а с процессором Pentium - от 20 и более. Эти цифры подтверждаются опытом реальной эксплуатации систем на основе соответствующих процессоров.

В ``персональных" операционных системах, таких как Windows 3.x и Windows 95, ситуация с памятью, и вообще с производительностью, несколько иная.

Дело в том, что пользователя ПК интересует не производительность в целом, а только время реакции текстового редактора или электронных таблиц на нажатие кнопки. С одной стороны, это вынуждает устанавливать в систему больше памяти, потому что страничный обмен во время отработки команды в редакторе существенно снижает время реакции и, соответственно, наблюдаемую скорость системы. Поэтому память часто подбирают так, чтобы используемые программы и их данные входили в память целиком, а не только рабочие множества. Но, так как пользователь обычно работает только с одной программой, часто оказывается возможным ограничиться 4 или 8 мегабайтами памяти. С другой стороны, низкокачественные прикладные программы часто вынуждают пользователей покупать чрезмерно мощные процессоры, чтобы обеспечить приемлемое время реакции. Так например, редактор MS Word 6.0 требует процессора класса i486/Pentium для редактирования текстов - работы, с которой вполне справляется даже 8-разрядный процессор типа Z-80.

Поэтому во многих конторах часто можно увидеть машину класса Pentium с 8 мегабайтами памяти, используемую только для печати бумажек, то есть очень сильно разбалансированную и очень сильно недогруженную по стандартам систем общего назначения.

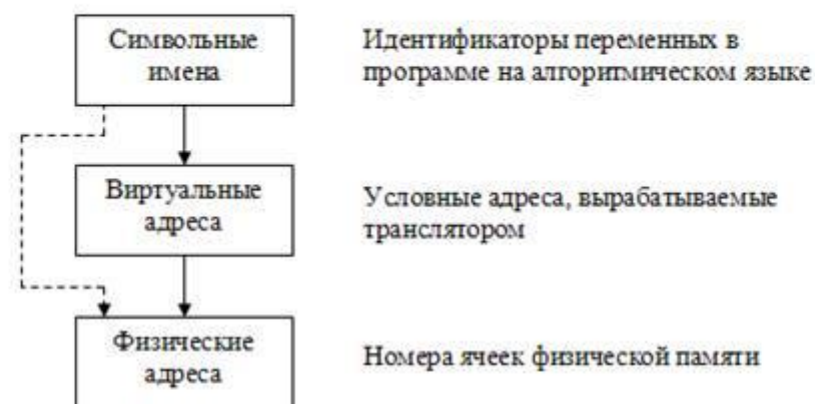
Виртуальная память

Одним из наиболее популярных способов управления памятью в современных ОС является так называемая **виртуальная память**. Наличие в ОС **виртуальной памяти**

позволяет программисту писать программу так, как будто в его распоряжении имеется однородная ОП большого объема, часто существенно превышающего объем имеющейся физической памяти. В действительности все данные, используемые программой, хранятся на диске и при необходимости частями (сегментами или страницами) отображаются в физическую память. При перемещении кодов и данных между ОП и диском подсистема ВП выполняет трансляцию виртуальных адресов, полученных в результате компиляции и компоновки программы, в физические адреса ячеек ОП.

Для идентификации переменных и команд на разных этапах цикла программы используются символьные номера (метки), виртуальные адреса и физические адреса:

Типы адресов



Символьные номера присваивает пользователь при написании программы на алгоритмическом языке или ассемблере (имена переменных и входных точек программных модулей).

Физические адреса соответствуют номерам ячеек ОП, где в действительности будут расположены переменные и команды. Физическая память представляет собой упорядоченное множество ячеек, и все они пронумерованы, т.е. к каждой из них можно обратиться, указав ее порядковый номер (адрес). Количество ячеек физической памяти ограниченно и фиксировано.

Виртуальные адреса (математические или логические), вырабатывает транслятор, переводящий программу на математический язык. Поскольку во время трансляции в общем случае неизвестно в какое место ОП будет загружена программа, то транслятор присваивает переменным и командам виртуальные (условные) адреса, обычно считая по умолчанию, что начальным адресом будет нулевой адрес.

ОС должна связать каждое указанное пользователем имя с физической ячейкой памяти, т.е. осуществить отображение пространства имен на физическую память компьютера. В общем случае это отображение осуществляется в два этапа: сначала системой программирования, а затем ОС (с помощью специальных программных модулей

управления памятью и использования соответствующих аппаратных средств вычислительной системы). Между этими этапами обращения к памяти имеют форму **виртуального адреса**.

Переход от **виртуальных адресов** к физическим может осуществляться двумя способами. В первом случае замену виртуальных адресов на физические делает специальная системная программа - перемещающий загрузчик. Перемещающий загрузчик на основании имеющихся у него исходных данных о начальном адресе физической памяти, в которую предстоит загружать программу, и информации, предоставленной транслятором об адресно-зависимых константах программы, выполняет загрузку программы, совмещая ее с заменой виртуальных адресов физическими.

Второй способ заключается в том, что программа загружается в память в неизмененном виде в виртуальных адресах, при этом операционная система фиксирует смещение действительного расположения программного кода относительно **виртуального адресного пространства**. Во время выполнения программы при каждом обращении к **оперативной памяти** выполняется преобразование виртуального адреса в физический. Второй способ является более гибким, он допускает перемещение программы во время ее выполнения, в то время как перемещающий загрузчик жестко привязывает программу к первоначально выделенному ей участку памяти. Вместе с тем использование перемещающего загрузчика уменьшает накладные расходы, так как преобразование каждого виртуального адреса происходит только один раз во время загрузки, а во втором случае - каждый раз при обращении по данному адресу.

В некоторых случаях (обычно в специализированных системах), когда заранее точно известно, в какой области **оперативной памяти** будет выполняться программа, транслятор выдает исполняемый код сразу в физических адресах.

Виртуальное адресное пространство

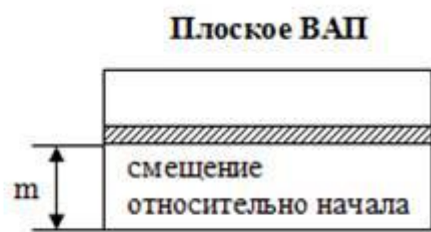
Совокупность *виртуальных адресов* (ВА) процесса называется **виртуальным адресным пространством** (ВАП). Диапазон возможных адресов виртуального пространства у всех процессов является одним и тем же.

При использовании 32-разрядных виртуальных адресов диапазон возможных адресов виртуального пространства задается границами 0000000016 и FFFFFFFF16.

Каждый процесс имеет собственное ВАП – транслятор присваивает виртуальные адреса переменным и кодам каждой программе независимо.

Совпадение **виртуальных адресов** и команд различных процессов не приводит к конфликтам, т.к. в том случае, когда эти переменные одновременно присутствуют в памяти, ОС отображает их на разные физические адреса. В разных ОС используются разные способы структуризации ВАП.

В одних ОС ВАП процесса подобно физической памяти представлены в виде непрерывной **линейной последовательности** виртуальных адресов. Такую структуру адресного пространства называют также **плоской** (flat). При этом виртуальным является единственное число, представляющее собой смещение относительно начала (обычно это значение 000…000) ВАП. Адрес такого типа называют линейным.



В других ОС ВАП делится на части, называемые сегментами. В этом случае помимо линейного адреса может быть использован виртуальный адрес, представляющий собой пару чисел (n, m), где n определяет сегмент, а m – смещение внутри сегмента. Существуют и более сложные способы структуризации ВАП, когда виртуальные адреса образуется тремя и более числами. Задачей ОС является отображение индивидуальных ВАП всех одновременно выполняющихся процессов физическую память.

Необходимо различать максимально возможное ВАП процесса и назначенное (выделенное) процессу ВАП. В первом случае речь идет о максимальном размере ВАП, определяемом архитектурой компьютера, на котором работает ОС, и, в частности, разрядностью его схем адресации (32-битная, 64-битная и т.п.), например, при работе на компьютерах с 32-разрядными процессорами Intel Pentium ОС может предоставить каждому процессу ВАП до 4 Гбайт.

Назначенная ВАП представляет собой набор виртуальных адресов, действительно нужных процессу для работы. Эти адреса первоначально назначает программе транслятор на основании текста программы, когда создает кодовый сегмент, а также сегмент или сегменты данных, с которыми программа работает. Затем при создании процесса ОС фиксирует назначенное ВАП в своих системных таблицах. В ходе своего выполнения процесс может увеличить размер первоначального, назначенного ему ВАП, запросив у ОС создание дополнительных сегментов и увеличение размера существующих.

Максимальный размер ВАП ограничивается только разрядностью адреса, присущей данной архитектуре компьютера и, как правило, не совпадает с объемом физической памяти, имеющейся в компьютере.

<u>Порядок выполнения работы.</u>

1. Ознакомиться с теоретическими сведениями о способах структуризации ОП.
2. Практически управлять ОП.
3. Сделать отчет по способам организации и управления ОП.
4. Пройти тестирование на компьютере по теоретическому материалу.

Требования к отчету.

Отчет должен включать :

- название работы и ее цель;
- результаты основных операций управления оперативной памяти;
- иллюстрации с примерами параметров и настроек реестра.

Список литературы.

1. Таненбаум Э., Современные операционные системы. 3-е издание. – Питер, 2010 – 1116с.
2. Таллоч М., Нортроп Т., Ханикатт Дж., Вилсон Э. Ресурсы Windows 7. – BHV Русская редакция, 2011 – 1104с.
3. Леонов В. Команды Linux. – Эксмо, 2011 – 176с.
4. Коварт Р., Уотерс Б. Windows NT Server4. – Питер, 2000- 448с.