## Классификация способов управления ОЗУ

В первых вариантах компьютеров, *ОС как таковые — отсутствовали*:

* + - *программа пользователя* загружалась в ОЗУ с адреса, который указывал оператор ЭВМ;
    - *после загрузки* программы, оператор указывал на пульте ЭВМ стартовый адрес программы и нажимал кнопку «Пуск»;
    - *после выполнения* текущей программы, в ЭВМ загружадась новая программа. Фактически, в таком (*пакетном*) режиме работы ЭВМ, управление памятью не проводилось или проводилось в упрощенных вариантах.

С появлением и совершенствованием первых ОС, которые также, как и прик- ладное ПО, являются программным обеспечением, стали применять *три способа загрузки*,

*Развите этих способов* загрузки ПО показано слева на право:

* + - *певый способ* предполагает первоначальную загрузку ОС, обычно в младшие адреса ОЗУ; такой способ применялся в *мэйнфреймах* и первых миником-

пьютерах; программы пользователя загружались с помощью ПО ОС в ос- тальную часть ОЗУ;

* + - *второй способ* стал применяться в некоторых карманных компьютерах и

*встроенных системах*, которые обладают малыми ресурсами и, как правило,

разными конструктивными особенностями; здесь ОС пишется в ПЗУ, которое соответствует верхним адресам памяти, оставляя ОЗУ полностью для ПО пользователей;

* + - *третий способ* соответствует *большинству современных систем*, подобных IBM PC; в ПЗУ записывается ПО BIOS, которое работает только при включе-

нии питания компьютера; ПО BIOS находит и загружает в младшие адреса ОЗУ ПО ОС; программам пользователей предоставляется оставшееся ОЗУ *до 640 Кбайт*, в младших адресах, и вся оставшаяся память *свыше первого Мбайт*; память ОЗУ между 640 Кб и 1 Мб (*скрытая область*) распреде- ляется между ПО BIOS и памятью прямого доступа для внешних устройств.

Для компьютера, память (ОЗУ) всегда была и остается *дефицитом*:

* + - *всегда имеются программы*, которым, по тем или иным причинам, недоста- точно выделенного объема ОЗУ ЭВМ;
    - *проблемы реализации языков программирования* требуют решать многие воп- росы, связанные с адресацией памяти и ее распределением.

Для решения этих и многих других вопросов было введено *понятие виртуальной памяти*.

**Виртуальная память**, призвана *разрешить противоречие в адресации команд и данных*, связанных с потребностями прикладного ПО и ресурсами ОЗУ, которые имеет конкретная ЭВМ.

Идея состоит в использовании *таблицы виртуальных адресов*, которая указы- вает адресам всех команд и данных, выставленных компилятором, соответствие физических адресов ОЗУ.

Реализация такого соответствия может выполняться:

* + - специальным загрузчиком (*статически*);
    - или специальными аппаратными средствами (*динамически*).

Появление ОС, поддерживающих *мультипрограммирование*, потребовало ре- шения вопросов распределения памяти ОЗУ между многими одновременно выпол- няемыми программами (*процессами*). На рисунке 1.5, приведена классификация различных подходов, решающих поставленную проблему.

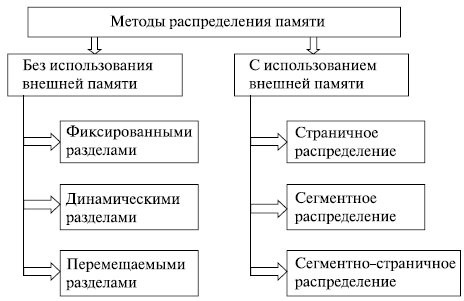


Рисунок 1.5 - Классификация способов распределения памяти Здесь можно выделить две большие группы подходов:

* + - методы, *не использующие внешнюю память*, обычно, накопители на жестких дисках;
    - методы, *использующие внешнюю память*, которая теоретически продолжает (расширяет, увеличивает) ОЗУ ЭВМ.

*Методы, не использующие* внешнюю память, предполагают, что *ОЗУ доста- точно для решения самой крупной задачи*, или, что программу можно исполнять по частям, для которых ОЗУ достаточно. Здесь, в большинстве случаев, задачи распре- деления памяти ставятся как *оптимальное разбиение ОЗУ на сегменты* (*отдельные и не пересекающиеся участки памяти*), обеспечивающие максимальную эффектив- ность использования процессорного времени ЭВМ.

*Методы, использующие* внешнюю память, предполагают, что ОЗУ расширя- ется за счет памяти медленных внешних устройств и становится возможным однов- ременное выполнение необходимого числа «параллельно» работающих процессов.

Здесь, в большинстве случаев, задачи распределения памяти ставятся как ограниче- ния на скорость выполнения отдельных процессов или эффективность использова- ния ОЗУ ЭВМ по управлению самим процессом распределения памяти ЭВМ.

В любом из указанных случаев, ОЗУ расходуется не идеально, а эффектив- ность используемых идей сильно зависит как от величины соотношения *объем ОЗУ/размер процессов*, так и от *аппаратной реализации поддержки* указанных подходов.

**Основными причинами** неэффективности использования ОЗУ являются:

* + - *сильная фрагментация памяти* как во время загрузки, так и во время завер- шения процессов;
    - *затраты, связанные с устранением последствий* такой фрагментации.

**Замечание** Фрагментация вызывается не только из-за различных размеров запускаемых про- цессов, но и, по-причине, *дополнительных требований ряда процессов к динами- ческому выделению дополнительной памяти ОЗУ*.

**В целом**, можно выделить ряд общих задач, по распределению памяти ЭВМ, которые должны решаться каждой ОС:

* + - отслеживание (учет) свободной и занятой памяти;
    - первоначальное и динамическое выделение памяти процессам приложений и самой операционной системе, а также освобождение памяти по завершении

процессов;

* + - настройка адресов программы на конкретную область физической памяти;
    - полное или частичное вытеснение кодов и данных процессов из ОЗУ на диск, когда размеры ОЗУ недостаточны для размещения всех процессов, а также

возвращение этих процессов в ОЗУ;

* + - защита памяти, выделенной процессу, от возможных вмешательств со сторо- ны других процессов;
    - дефрагментация памяти.

Решение всех указанных задач возложено на «*Подсистему распределения памяти*»,

которая, в свою очередь, входит в «*Подсистему управления процессами*» ядра ОС.

## Программный и аппаратный способы адресации памяти

Нормальная работа компьютера обеспечивается двумя способами адресации памяти ЭВМ: *программным* и *аппаратным*.

*Программный способ адресации неявно присутствует* в любом исходном тексте программы в виде операторов команд программы, которые обращаются к объявленным переменным данных и осуществляют различные управляющие дейст- вия по определению порядка выполнения самих команд. В последствии, компиля- тор обеспечит *явное присутствие* как адресов команд, так и адресов данных.

Соответственно, в каждом отдельном процессе, *минимально* можно выде-

лить:

* + - управляющий блок процесса;
    - сегмент кода (программы);
    - сегмент данных;
    - стек;
    - «куча», если процесс использует динамическое выделение памяти.

Аппаратный способ адресации заложен сам процессор:

* + - *специализацией регистров процессора*: сегментные, смещения и индексные;
    - *методами адресации команд процессора*;
    - *шинами* (каналами) *процессора*.

Реализация такой схемы имеет *недостатки*, поскольку:

* + - *каждым, из выделенных блоков*, необходимо управлять;
    - *каждый, из выделенных блоков*, использует относительную адресацию, кото- рая требует использования сумматоров для каждой команды и данных про-

цесса;

* + - *каждая команда или обращение данных* требуют проверки на возможность нарушения границ, выделенных областей.

Таким образом, эффективность использования процессора значительно снижается, по-причине использования *сложной системы адресации памяти*.

Чтобы устранить эти недостатки, во всех разви-тых архитектурах ЭВМ используются *аппаратные средства виртуализации памяти* (***MMU***). Это освобождает процессор от указанной выше рутинной работы.

**MMU** — *Memory Managment Unit* — *диспетчер памяти* уже был рассмотрен ранее, при анализе прямого доступа к памяти внешних устройств. В его обязан- ности входит:

* + - виртуализация адреса, при взаимодействии процессора с ОЗУ;
    - аппаратное обеспечение алгоритмов распределения блоков ОЗУ в различных режмах адресации.

## Страничная и сегментная адресации памяти

Исторически, для управления памятью ОЗУ ЭВМ, в ОС стала разрабаты- ваться *сегментная адресация*.

Сегментная адресация ОЗУ реализуется с помощью *специальных регистров* процессора, которые так и называются: *регистры сегментов* и *регистры смещения*. *Сегментный способ адресации ОЗУ* всегда использует таблицу сегментов, которая сама размещается в ОЗУ и предназначена для решения двух основных

задач:

* + - виртуализация относительных адресов программы относительно физической памяти ЭВМ;
    - управление свопингом и перемещение сегментов в физическом пространстве ОЗУ.

**Задачи виртуализации памяти** могут решаться как с привлечением, так и без привлечения специальных аппаратных средств (***MMU***).

**Управление свопингом** — управление перемещением сегментов программ из ОЗУ в специальные файлы или разделы свопинга, расположенные на внешних ЗУ, и обратно.

Для этой цели, таблица сегментов должна содержит управляющую информа- цию, которая указывает не только на наличие или отсутствие сегментв в ОЗУ, но и используется для целей защиты или разделения памяти между процессами.

**Замечание** *Основная проблема* сегментгой адресации ОЗУ заключается в ее *фрагментации*, кото- рая вызвана *разным размером* загружаемых и выгружаемых сегментов:

* + - *при равных размерах* сегментов, фрагментация была бы минимальна или ее удалось бы устранить;
    - *чем меньше разделы сегментов*, тем меньше становятся непроизводительные потери ОЗУ.

Идея использования блоков одинакового размера, для адресации физической и логической частей ОЗУ, стала называться *страничной организацией памяти*.

*Основное достоинство* страничной адресации ОЗУ - *отсутствие фрагмен- тации*, что не требует затрат ресурса ЭВМ на ее устранение.

схема адресации ОЗУ с помощью страниц очень похожа на схему адресации ОЗУ с помощью сегментов, поскольку в обоих случаях используются таблицы переадресации, которые должны размещаться в той же ОЗУ.

**Замечание** В связи с малым размером страницы, обычно 4 Кбайт, *увеличивается скорость пэйджинга*: перемещения страницы на внешний носитель ЗУ и обратно.

**Пэйджинг** (*pagging*) - свопинг (*swapping*) страниц.

**По традиции**, файлы или разделы винчестера, *куда осуществляется pagging*, называ- ются файлами или разделами свопинга.

**Очевидно**, что аппратные средства виртуализации (***MMU***) способны обеспечивать как

*pagging*, так и *swapping*.

*Основная проблема* адресации с помощью страниц - *большой размер таблиц адресации*, которые должны храниться в ОЗУ.

Чтобы частично устранить эту проблему, стали использовать *двухуровневую схему страничной адресации*, ставшую основой адресации 32-битных ЭВМ:

* + - *в ОЗУ постоянно находится* только корневая таблица страниц, содержащая не более 1024 записи;
    - *когда программа загружается* на выполнение, в корневую таблицу страниц заностися запись о местоположении и числе требуемых ей страниц;
    - *внутри самой программы*, создается своя собственная таблица, которая заполняется во время загрузки программы и удаляется после ее завершения.

*Другая проблема*, с которой сталкиется страничный способ адресации — значительные затраты времени, связанные с поиском информации о странице в больших таблицах страниц.

Чтобы разрешить эту проблему, стали использовать *таблицу быстрой транс- ляции адресов* или — *буфер TLB*.

**TLB** — *Translation Lookaside Buffer* — ассоциативная кэш-память, в которую записывается реально используемая страница, перед тем как команды и данные будет использовать сам процессор.

Алгоритм использования TLB — прост:

* + - когда процессор обращается к странице и *она присутствует в TLB*, то используется команды и данные, считанные из нее;
    - когда процессор обращается к странице и *она отсутствует в TLB*, возникает прерывание к соответствующей программе ядра ОС;
    - обработка прерываний осуществляется «*Подсистемой управления памятью ОС*», которая производит поиск нужной страницы в ОЗУ или на внешнем

носителе, решает вопрос об удалении ненужной страницы и записи нужной, перезапускает команду процессора, вызвавшую прерывание по отсутствию страницы в TLB.

### Замечание

К сожалению, проблема управления страницами не решается так просто, например, для 64-разрядных ЭВМ создание таблицы страниц становится *не реализуемым*.

Чтобы теоретически закрыть изучаемую тему, рассмотрим комбинированный способ адресации.

## Комбинированный способ адресации памяти

Преимущества адресации имеются у каждого способа:

* + - сегментная адресация памяти — требует относительно малое число записей в таблице сегментов.
    - страничная адресация памяти — устраняет проблемы фрагментации, пос- кольку ОЗУ адресуется с помощью *произвольного доступа*, следовательно, свободную страницу ОЗУ можно найти в любом месте памяти.

Отсюда возникает *идея смешанной (комбинированной) адресации*, которая состоит в том, что:

* + - *отдельная программа* (процесс) отображается в виде нескольких записей в общей таблице сегментов;
    - *кажный сегмент*, выделенный программе, отображается внутри нее в виде

одной или нескольких таблиц страниц.

**Таким образом**, сегментно-страничная адресация памяти похожа на *подоб- ную двухуровневую страничную адресацию*. При этом:

* + - *существенно уменьшается размер* корневой таблицы, постоянно хранящейся в ОЗУ и как следствие ускоряется поиск в ней нужной страницы;
    - *трехкомпонентная адресация* (номер сегмента, номер страницы, смещение) требует аппаратной поддержки виртуализации памяти с помощью MMU, без которой она становится не эффективной;
    - *появляется возможность оптимизации* схем адресации на базе учета разных свойств разных сегментов программы; например, *сегменты кода предназна- чены только для чтения*.

**В общем случае**, применяемая схема адресации ОЗУ зависит не только от конструктивных возможностей физической памяти или аппаратной поддержки вируализации памяти с помощью устройств MMU, но и от *конструктивных особенностей самого процессора*.

*Такая зависимость*, стала поддерживаться процессорами компании Intel, начиная с архитектуры i386. Здесь выделяются:

* + - глобальная таблица дескрипторов (***GDT***), на которую должен указывать

специальный регистр процессора — ***GDTR***;

* + - локальные таблицы дескрипторов (***LDT***), на которые должен указывать специальный регистр процессора — ***LDTR***.

*Глобальная дескрипторная таблица* является единственной и контролизуется ядром ОС, адресуя сегменты процесса и значения регистра LDTR.

*Локальные дескрипторные таблицы* связаны с конкретными испольняемыми процессами и формируются при их запуске.

Указанная схема адресации, широко используется в ОС MS Windows, по крайней мере, начиная с Windows 2000, выполненной по технологии NT.

**Замечание** В качестве указателей на GDT и LDT, используются *16-разрядные слова*, в которых младшие три бита имеют специальное значение:

* + - *0-й и 1-й биты* — уровень защиты (уровень привилегий) работающей задачи;
    - *2-й бит* используется для различения GDT и LDT.

следует понимать, что *ОС «скрывает»* от процессов детали способов адресации памяти, возлагая реализацию ее распределения на «*Подсистему управления памятью*».

## Системные вызовы ОС по управлению памятью

## Структура поцесса

Как было отмечено ранее, основная память (ОП) или ее рассматриваемя часть ОЗУ занимает особое положение в плане теории и практики ее применения.

Причина такой особоенности состоит в том, что для управления чем-либо, включая распределение ОЗУ, необходимо программное обеспечение, которое само должно быть загружено в ОЗУ. Поэтому для программ (процессов), выполняющих- ся в пользовательском режиме, многие аспекты распределения памяти или скрыты совсем, или проявляются только косвенно.

*Виртуальная структура процесса* состоит из отдельных частей, которые, по- традиции, также называются сегментами:

* *cегмент text* - содержит машинные инструкции и константы; его содержание не изменяется при выполнения процесса;
* *сегмент data* - содержит внешние и статические переменные, *которые ини- циализированы*, при компиляции программы процесса;
* *сегмент bss* - содержит внешние и статические переменные, *которые неини- циализированы*, при компиляции программы процесса;
* *сегмент stack* - используется для хранения значений автоматических пере- менных и параметров функций, а также для организации рекурсивных вызовов функций.

**Замечание**

Между нижней границей стека и сегментов данных обычно существует *адресный зазор*, который может быть использован при выполнении процесса для расширения сегментов стека и данных. Такое расширение возможно, пока не исчерпано пространство свобод- ных адресов.

Когда процесс создан, то измение размера его сегментов осуществляется в случаях загрузки *нового тела процесса*. Если новое тело не загружается, то:

* изменение размера *сегмента кода процесса* не происходит, поскольку этот сегмент предназначен только для чтения;
* изменение размера *сегментов данных* осуществляется за счет сегмента BSS и, обычно, реализуется системными вызовами динамического распределения памяти *brk(...)* и *sbrk(...)*;
* изменение размеров *сегментов стеков* осуществляется автоматически, во время выполнения процесса.

**Замечание**

*Считается*, что ОС Linux придерживается идеологии ОС UNIX и, поэтому, должна иметь

одинаковую структуру сегментов процесса.

*На самом деле*, - это не так, хотя общая идея выделения *сегмента кода* и *сегмента данных + стека* — присутствует.

В общем случае, учесть все детали распределения сегментов процессов — довольно проблематично, тем более, что эти детали могут меняться от одной ОС к другой.

## Определяемые сегменты процесса

Как отмечено ранее, любой программе пользователя для своей работы необ- ходима память ОЗУ. Память ОЗУ выделяется программе в виде сегментов или в виде страниц, или другим комбинированным способом. Все разнообразие способов выделения памяти программы учитывается в ядре ОС и доступна программистам, занимающимся разработкой ПО ядра. Кроме того, имеются три параметра любого процесса, которые доступны любой программе на языке ***С***. Такими параметрами (*внешними переменными*) являются:

* *etext* — первый адрес, следующий за последним адресом сегмента кода про- цесса (*text segment*);
* *edata* — первый адрес, следующий за последним адресом сегмента иници- ализированных данных процесса (*initialized data segment*);
* *end* — первый адрес, следующий за последним адресом сегмента не иници- ализированных данных процесса (*uninitialized data segment*), известный также как *BSS*.

Чтобы подробно изучить назначение описанных выше внешних переменных, сле- дует воспользоваться командой: ***man etext***. Там же, приведены примеры их исполь- зования.

**Замечание**

Естественно, что стандартные средства языка ***С*** не отражают многих характеристик реальных процессов ОС, поскольку эти характеристики связаны с особенностями реализации конкретных систем.

В следующем подразделе, мы рассмотрим характеристики процессов 64-битных ОС Linux, которой также является ОС УПК АСУ.

Значения многих параметров процесса можно посмотреть непосредственно, используя возможности «просмотрищика» файлого менеджера «*Midnight Comman- der*».

## Создание и удаление процессов из памяти

Структура процессов ОС Linux следует современным тенденциям построе- ния операционных систем, в плане использования ими основной памяти ЭВМ:

* *они полностью используют* виртуальную адресацию памяти;
* *ядро ОС фиксирует* основные характеристики всех запущенных процессов.

В настоящее время, ядро ОС использует *32-битную виртуальную адресацию*

пространства процесса, которая составляет размер в 4 ГБ, как показано на рисунке 2.3:

* *первые 3 ГБ* соответствуют пространству пользователя (*User space*);
* *последний 1 ГБ* находится в распоряжении ядра ОС (*Kernel space*).

В ядре ОС, для каждого процесса отведена *директория страниц*, что подра- зумевает возможность доступа к *1КВ таблицам страниц*, которые указывают на *1МВ 4-х килобайтных страниц*. Это позволяет адресовать 4 ГБ памяти.

Каждый пользовательский процесс имеет свою *локальную таблицу дескрип- тора*, которая адресует различные *сегменты кода* и *сегменты данные-стек*.

В пользовательском пространстве, линейные адреса и логические адреса идентичны.

Процесс получает свои таблицы страниц от родителя, при выполнении пос- ледним системного вызова *fork()*, со входами, помеченными как *READ-ONLY* или *замещаемые*.

Если процесс пытается писать в некоторую область памяти и страница явля- ется *COPY-ON-WRITE* страницей, то она копируется и помечается как *READ- WRITE*.

В любом случае, «*Cистема управления памятью*» Linux осуществляет *подкачку страниц по обращению*, в соответствии со стратегией *COPY-ON-WRITE*, которая *основана на механизме подкачки* и поддерживается процессором i386.

Современные ядра ОС Linux псевдоустройство *proc*, с файловой системой типа *proc*, которая монтируется к директории /proc корневой ФС.

Эта файловая система содержит много информации о каждом запущенном процессе:

* директория */proc/<номер процесса>* - содержит файлы (псевдофайлы) о раз- ных характеристиках процесса с конкретным номером ***PID***;
* директория */proc/self* — является уникальной ссылкой для процесса, который обращается к информации о себе и который, как любой дочерний процесс, не знает свой ***PID***.
  1. **Динамическое выделение и освобождение памяти**

**процесса**

В предыдущих подразделах были рассмотрены случаи выделения памяти, которые осуществляются ядром ОС.

В данном подразделе, мы изучим системные вызовы ОС, обеспечивающие выделение памяти для нужд программиста. Для этих целей существует ряд функ- ций, позволяющих *выделять и освобождать память ОЗУ динамически*, во время выполнения программы.

Семантика этих функций - следующая:

* *malloc(...)* - выделяет *область неинициализированной памяти*, размер которой указан в байтах, в качестве аргумента, и возвращает значение указателя на

эту область; если значение аргумента равно нулю, то возвращается значение указателя *NULL*;

* *free(...)* — освобождает память ОЗУ, которая *ранее была выделена* одной из функций: *malloc(...)*, *calloc(...)* или *realoc(...)*; если аргумент функции равен

*NULL*, то никаких действий не производится;

* *calloc(...)* - выделяет память ОЗУ *под массив данных*, размерность которого задана первым аргументом, а размер каждого элемента массива в байтах,

задан вторым аргументом; при этом, значение каждого байта выделенной памяти обнуляется; возвращает указатель на выделенный массив данных; если значение любого из аргументов функции равно нулю, то память ОЗУ процессу не выделяется и возвращается значение *NULL*;

* *realloc(...)* - изменяет *размер уже выделенного блока памяти*, используя пер- вый аргумент как указатель, а второй — как нужный размер; если размер

области уменьшается, то значения байт оставшейся области не изменяется; если размер области увеличивается, то добавляемая область не инициали- зируетсся; если значение аргумента указателя равно *NULL*, то результат функции, соответствует действию функции *malloc(...)*; если требование на новый размер памяти равно нулю, а указатель не равен *NULL*, то действие эквивалентно функции *free(...)*.

Кроме перечисленных, имеются также системные функции *brk(...)* и *sbrk(...)*, с помощью которых можно непосредственно изменять *конец сегмента неинициа- лизированных данных*.

Семантика системных вызовов:

* *brk(...)* - устанавливает конец сегмента данных в значение, указанное в аргу- менте *addr*, когда это значение является приемлимым; указание адреса памя- ти должно быть в пределах максимально возможного размера сегмента дан-

ных (см. *setrlimit(2)*);

* *sbrk(...)* - увеличивает пространство данных программы на *increment* байт; *sbrk(...)* не является системным вызовом, он просто является оберткой (wrapper), которую использует библиотека ***C*** для вызова *brk(...)*; вызов

*sbrk(...)* с инкрементом *0* может быть использован, чтобы найти текущее местоположение прерывания программы.

В случае успеха *brk(...)* возвращает ноль, а *sbrk(...)* возвращает указатель *на начало новой области*. В случае ошибки, возвращается *-1* и значение *errno* устанавли- вается в значение *ENOMEM*.