Адресные пространства процессора

Все дело во взаимосвязи и сути трех наиважнейших понятий в идеологии адресации: виртуальное, линейное и физическое адресные пространства. Все эти понятия - отдельные сущности, взаимодействуют между собой, но ни в коем случае нельзя путать эти термины, поскольку на их совокупности зиждется любая ОС.

Например, очень часто в литературе по ОС можно встретить такие высказывания: "виртуальное (линейное) адресное пространство процесса..." или некоторые даже не указывают, какое именно адресное пространство имеется ввиду - просто "адресное пространство процесса..."

Виртуальное адресное пространство процессора Intel (IA32) в защищенном режиме

Виртуальный адрес (он же логический) в процессорах с 32-х разрядной архитектурой (IA32) - совокупность всех возможных адресов вида:

сегментный_регистр:смещение

Например, CS:12345678h - виртуальный (логический) адрес

В старой официальной документации (1995 г.) адреса такого вида назывались интеловцами виртуальными, в новой (2002 г.) они дают новое определение - логический, но по сути - это синонимы.

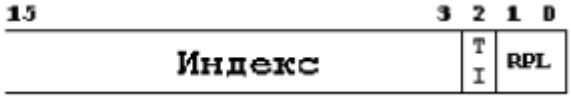
Почему виртуальный? Где здесь виртуальная сущность? Виртуальная сущность в том и состоит, что глядя на нечто вроде 6666h:12345678h никогда нельзя сказать наверняка, однозначно, что на самом деле представляют из себя эти два числа. Знакомые с защищенным режимом сразу определят, что это некий адрес, и по левой части (селектору) можно найти число (базу), прибавляя к которой правую часть получится некий другой адрес, в общем, 6666h:12345678h - это некий виртуальный адрес, на самом деле (физически) его и нет вовсе...

Теперь давайте разберемся, каков размер виртуального (логического) адресного пространства процессора.

Итак, рассмотрим правую часть виртуального адреса. Здесь все и так понятно: 2^32 = 4Гб, т.е. правая часть виртуального адреса может принять 4Гб различных значений.

Но есть еще и левая часть виртуального адреса, о которой почему-то авторы многих книг забывают.

Селектор:



Селектор - это и есть та самая левая часть виртуального адреса.

Теперь, избавимся от битов, которые не отвечают за адресацию. Это два бита RPL. В теории можно подмахнуть сюда еще и TI, но этого делать не нужно, т.к. TI - это индикатор таблицы дескрипторов, и он еще как отвечает за адрес, который в конце концов вылезет на адресную шину...

Теперь:

GDT может содержать 8192 дескриптора.

LDT может содержать 8192 дескриптора.

Нулевой дескриптор в GDT - недоступен.

Значит, имеем 8191 - в GDT и 8192 - в LDT.

Всего - 16383 дескрипторов. И каждый - 4Гб.

Всего: 16383 * 4Гб = 65532 Гб (64 Тб – 4 Гб)

Иначе говоря, виртуальный адрес - не 32-х разрядный.

Он 46-разрядный. (32 бита смещения + 1 бит (TI) + 13 бит (индекс))

2^46 байт = 65536Гб = 64 терабайт (-4 Гб) - это и есть виртуальное адресное пространство процессора Интел в защищенном режиме.

(но реально, за вычетом нулевого дескриптора из GDT, имеем на 4Гб меньше адресов - 65532Гб)

Линейное адресное пространство процессора Intel (IA32) в защищенном режиме

Линейный адрес - это 32-х разрядный адрес, получаемый путем прибавления 32-х разрядного смещения (правая часть виртуального адреса) к базе сегмента.

Линейный адрес - всегда 32-х разряден. В этом легко убедиться, попытавшись опровергнуть это утверждение. Попробуйте создать сегмент с базой в 4Гб и сделать что-то типа jmp селектор_этого_сегмента:2Гб (т.е. фактически должен возникнуть линейный адрес = 4 Гб (база) + 2 Гб (смещение) = 6 Гб). Вместо этого мы получим #GP. Вот и все.

Поэтому, линейное адресное пространство процессора Intel в защищенном режиме составляет 4 Гб.
Физическое адресное пространство процессора Intel (IA32) в защищенном режиме
Здесь возможны два варианта:
В режиме сегментной адресации физический адрес всегда совпадает с линейным.
В режиме сегментно-страничной адресации физический адрес не совпадает с линейным, а получается путем разбиения линейного адреса на 3 части и путешествия по каталогам и таблицам страниц.
По сути, при включенной сегментно-страничной адресации, линейный адрес также таит в себе некую "виртуальную сущность", потому что в принципе может быть представлен в виде X:Y:Z (действительно, он состоит из трех совершенно разных чисел: номера записи в каталоге страниц, номера записи в таблице страниц и смещения в странице). Но в это не стоит углубляться.
Физический адрес - это тот адрес, который процессор выставляет на адресную шину процессора.
Вот здесь возникает еще одно заблуждение: многие считают, что физический адрес и адрес в оперативной памяти - это одно и то же. На самом деле, если разобраться, физический адрес - может соответствовать адресу ячейки оперативной памяти (если она существует), либо не может (если ячейки по такому адресу не существует). В любом случае, следует помнить, что физический адрес - это не есть адрес ячейки оперативной памяти, хотя и может совпадать с ним.
Например, окончательно сформированный процессором из линейного адреса физический адрес составляет 678 Мб, он выставляется на адресную шину. А оперативной памяти на машине стоит, предположим, 256 Мб. Вот и ответьте сами себе: является ли выставленный физический адрес одновременно адресом ячейки в оперативной памяти? Конечно же нет, такой ячейки попросту не существует. Другой интересный вопрос - что произойдет в таком случае дальше? Здесь уже все зависит от чипсета материнской платы - либо процессору будет послан сигнал #RESET (что и происходит в 90% случаях), либо он просто получит фиктивные данные. Но это не суть важно - важно, что множество адресов ячеек оперативной памяти входит в множество физических адресов.
Виртуальное адресное пространство процессора Intel (IA32) в реальном режиме.
Оказывается, и в реальном режиме существует такое понятие.
В общем, все очень просто.
Виртуальный адрес в реальном режиме тоже состоит из двух частей:
сегмент:смещение
Максимальный виртуальный адрес в реальном режиме равен FFFFh:FFFFh, это очевидно. А сколько их всего, этих виртуальных адресов?
0000:0000
0000:0001
0000:0002
...
1234:5678
...
FFFF:FFFF
ровно FFFFFFFFh штук
а это есть 4Гб
Т.е. вывод: виртуальное адресное пространство процессора Интел в реальном режиме составляет 4 Гб.
Адрес 0000h:0010h и адрес 0001h:0000h преобразуются в один и тот же линейный адрес 00000010h. В один и тот же линейный адрес. Но фактически, на него указывают два разных виртуальных адреса.
Правило получения линейного адреса из виртуального в реальном режиме:
shl сегм.регистр, 4
add сегм.регистр, смещение
Пример 1:
1234h:5678h
Сдвинуть 1234h на 4 бита влево. Получаем 12340h.
Прибавляем 5678h. Получается: 12340h+5678h=179B8h
Т.е. вирт. адрес 1234h:5678h = лин. адресу 179B8h
Пример 2:
0001h:0000h
Сдвинуть 0001h на 4 бита влево. Получаем 0010h.
Прибавляем 0000h. Получается: 0010h+0000h=0010h
Т.е. вирт. адрес 0001h:0000h = лин. адресу 0010h
Всё в целом похоже на защищенный режим, только там вместо сдвига берется база и т.д.

Линейный адрес 00000010h можно получить только из двух разных виртуальных адресов. Но ведь, допустим, линейный адрес 00000020h можно получить уже из трех различных виртуальных адресов, а именно:
0000h:0020h
0001h:0010h
0002h:0000h
и все эти три виртуальных адреса - это линейный адрес 00000020h.
В реальном режиме линейный адрес всегда совпадает с физическим (выставляемым на адресную шину) (кстати, как и в случае сегментной адресации в защ. режиме).
Вывод: линейный адрес не совпадает с физическим только при страничной адресации в защ. режиме.
Перед тем, как рассмотреть переключение в защищённый режим, ещё раз освежим в памяти всё, что мы о нём уже узнали.
Механизм адресации: Так же, как и в реальном режиме, адрес складывается из адреса начала сегмента и относительного смещения, но если в реальном режиме адрес начала сегмента просто лежал в соответствующем сегментном регистре, деленный на 16, то в защищенном режиме не все так просто. В сегментных регистрах находятся специальные 16-битные структуры, называемые селекторами и имеющие следующий вид:
биты 15-3: номер дескриптора в таблице
бит 2: индикатор таблицы 0/1 - использовать GDT/LDT
биты 1-0: уровень привилегий запроса (RPL)
Уровень привилегий запроса — это число от 0 до 3, указывающее степень защиты сегмента, для доступа к которому применяется данный селектор. Если программа имеет более высокий уровень привилегий, при использовании этого сегмента привилегии понизятся до RPL. Уровни привилегий и весь механизм защиты в защищенном режиме нам для примера не потребуются.
GDT и LDT - таблицы глобальных и локальных дескрипторов соответственно. Это таблицы 8-байтных структур, называемых дескрипторами сегментов, где и находится начальный адрес сегмента вместе с другой важной информацией:
слово 3 (старшее):
биты 15-8: биты 31-24 базы
бит 7: бит гранулярности (0 - лимит в байтах, 1 - лимит в 4-килобайтных единицах)
бит 6: бит разрядности (0/1 - 16-битный/32-битный сегмент)
бит 5: 0
бит 4: зарезервировано для операционной системы
биты 3-0: биты 19-16 лимита

слово 2:
бит 15: бит присутствия сегмента
биты 14-13: уровень привилегий дескриптора (DPL)
бит 12: тип дескриптора (0 - системный, 1 - обычный)
биты 11-8: тип сегмента
биты 7-0: биты 23-16 базы

слово 1: биты 15-0 базы

слово 0 (младшее): биты 15-0 лимита

Два основных поля структуры, которые нам интересны, - это база и лимит сегмента. База представляет линейный 32-битный адрес начала сегмента, а лимит - 20-битное число, которое равно размеру сегмента в байтах (от 1 байта до 1 мегабайта), если бит гранулярности сброшен в ноль, или в единицах по 4096 байт (от 4 Кб до 4 Гб), если он установлен в 1. Числа отсчитываются от нуля, так что лимит 0 соответствует сегменту длиной 1 байт, точно так же, как база 0 соответствует первому байту памяти.
Остальные элементы дескриптора выполняют следующие функции:
1. Бит разрядности: для сегмента кода этот бит указывает на разрядность операндов и адресов по умолчанию. То есть в сегменте с этим битом, установленным в 1, все команды будут

интерпретироваться как 32-битные, а префиксы изменения разрядности адреса или операнда будут превращать их в 16-битные, и наоборот. Для сегментов данных бит разрядности управляет тем, какой регистр (SP или ESP) используют команды, работающие с этим сегментом данных как со стеком.
2. Поле DPL определяет уровень привилегий сегмента.
3. Бит присутствия указывает, что сегмент реально есть в памяти. Операционная система может выгрузить содержимое сегмента из памяти на диск и сбросить бит присутствия, а когда программа попытается к нему обратиться, произойдет исключение, обработчик которого снова загрузит содержимое данного сегмента в память.
4. Бит типа дескриптора - если он равен 1, сегмент является обычным сегментом кода или данных. Если этот бит - 0, дескриптор является одним из 16 возможных видов, определяемых полем типа сегмента.
5. Тип сегмента: для системных регистров в этом поле находится число:от 0 до 15, определяющее тип сегментов (LDT, TSS, различные шлюзы). Для обычных сегментов кода и данных эти четыре бита выполняют следующие функции:
бит 11: 0 - сегмент данных, 1 - сегмент кода
бит 10: для данных - бит направления роста сегмента
для кода - бит подчинения
бит 9: для данных - бит разрешения записи
для кода - бит разрешения чтения
бит 8: бит обращения
6. Бит обращения устанавливается в 1 при загрузке селектора этого сегмента в регистр.
7. Бит разрешения чтения/записи выбирает разрешаемые операции с сегментом - для сегмента кода это могут быть выполнение или выполнение/чтение, а для сегмента данных — чтение или чтение/запись.
8. Бит подчинения указывает, что данный сегмент кода является подчиненным. Это значит, что программа с низким уровнем привилегий может передать управление в сегмент кода и текущий уровень привилегий не изменится.
9. Бит направления роста, сегмента обращает смысл лимита сегмента. В сегментах с этим битом, сброшенным в ноль, разрешены абсолютно все смещения от 0 до лимита, а если этот бит 1, то допустимы все смещения, кроме от 0 до лимита. Про такой сегмент говорят, что он растет сверху вниз.
GDTR: 6-байтный регистр, в котором содержатся 32-битный линейный адрес начала таблицы глобальных дескрипторов (GDT) и ее 16-битный размер (минус 1). Каждый раз, когда происходит обращение к памяти, по селектору, находящемуся в сегментном регистре, определяется дескриптор из таблицы GDT или LDT, в котором записан адрес начала сегмента и другая информация.
LGDT источник
Команда загружает значение источника (6-байтная переменная в памяти) в регистр GDTR. Если текущая разрядность операндов 32 бита, в качестве размера таблицы глобальных дескрипторов используются младшие два байта операнда, а в качестве ее линейного адреса — следующие четыре. Если текущая разрядность операндов - 16 бит, для линейного адреса используются только байты 3, 4, 5 из операнда, а в самый старший байт адреса записываются нули.
Команда выполняется исключительно в реальном режиме или при CPL = 0.
SGDT приёмник
Помещает содержимое регистра GDTR в приемник (6-байтная переменная в памяти). Если текущая разрядность операндов - 16 бит, самый старший байт этой переменной заполняется нулями (начиная с 80386, а 286 заполнял его единицами).
LMSW источник/SMSW приёмник
LMSW копирует младшие четыре бита источника (16-битный регистр или переменная) в регистр CRO, изменяя биты PE, MP, EM и TS. Кроме того, если бит PE = 1, этой командой его нельзя обнулить, то есть нельзя выйти из защищенного режима. Команда LMSW существует только для совместимости с процессором 80286, и вместо нее всегда удобнее использовать mov cr0, eax.
Команда выполняется только в реальном режиме или с CPL = 0.
SMSW копирует младшие 16 бит регистра CRO в приемник (16- или 32-битный регистр или 16-битная переменная). Если приемник 32-битный, значения его старших битов не определены.

Команда SMSW нужна для совместимости в процессором 80286, и вместо нее удобнее использовать mov eax, cr0.
Переключение в защищенный режим
Для переключения в защищенный режим нужно выполнить ряд простых и незамысловатых по сути действий. Пока тех знаний, которыми мы обладаем, недостаточно для сложных задач, поэтому ограничимся малым - выведем на экран надпись.
Итак, для начала определимся с моделью памяти. Пусть это будет flat-модель (все сегменты имеют базу ноль и лимит 4 Гб). Рассмотрим сегментную адресацию.
Начать можно с чего угодно, как пример – с заполнения таблиц дескрипторов (дескриптор – описывает сегмент, селектор – указатель на дескриптор).
Таблица, которую нам нужно заполнить – таблица глобальных дескрипторов (GDT). Пусть в дескрипторах GDT будут описаны следующие сегменты:
сегмента кода (код, который будет исполняться в защ. режиме)
сегмент данных (данные, которые мы будем выводить на экран)
сегмент видеопамяти (фактически это сегмент, в который мы будем выводить наш текст)
Глобальная таблица дескрипторов:

```
GDT:
; нулевой дескриптор (обязательно должен присутствовать в GDT!)
NULL_descr db 8 dup (0)
; дескриптор 32-разрядного сегмента кода: база = 00000000h, размер = FFFFFFFFh
CODE_descr db 0FFh, 0FFh, 00h, 00h, 00h, 10011010b, 11001111b , 00h
; дескриптор 32-разрядного сегмента данных: база = 00000000h, размер = FFFFFFFFh
DATA_descr db 0FFh, 0FFh, 00h, 00h, 00h, 10010010b, 11001111b , 00h
; дескриптор сегмента видеопамяти: база = 000B8000h, размер = 0000FFFFh
VIDEO_descr db 0FFh, 0FFh, 00h, 80h, 0Bh, 10010010b, 01000000b , 00h
; размер таблицы GDT:
GDT_size db $-GDT
; а следующие три слова (размер GDT и линейный адрес начала таблицы) мы должны
будем попозже загрузить в GDTR:
GDTR dw GDT_size-1
dd ?
```

Вычислим линейный адрес таблицы GDT. Зачем? Чтоб загрузить регистр GDTR. Линейный адрес GDT = линейный адрес базы сегмента RM_CODE (потому что GDT расположена именно в этом сегменте у нас в программе) + смещение метки GDT в нем.

```
xor EAX,EAX ; обнуление регистра EAX
mov AX,RM_CODE ; теперь AX = номер сегмента RM_CODE
shl EAX,4 ; сдвигаем значение в EAX на 4 влево
; вот теперь EAX = линейный адрес базы сегмента RM_CODE
add AX,offset GDT ; теперь EAX = линейный адрес GDT
; линейный адрес GDT кладем в заранее подготовленную переменную:
mov dword ptr GDTR+2,EAX
; собственно, загрузка регистра GDTR:
lgdt fword ptr GDTR
```

Теперь нам еще нужно вычислить линейный адрес т.н. точки входа в защищенный режим. Дело в том, что после переключения в защ. режим мы окажемся в неком подвешенном состоянии – когда регистр CS еще содержит номер сегмента из реального режима, а не селектор. Мы не можем просто сделать
mov CS, селектор, CS можно изменить только дальним jmp либо iret. Нам все же придется делать дальний jmp для изменения CS. Но куда? Вот именно на точку входа в защищенный режим.

```
; вычисляем линейный адрес метки ENTRY_POINT (точка входа в защищенный режим):
xor EAX,EAX ; обнуляем регистра EAX
mov AX,PM_CODE ; AX = номер сегмента PM_CODE
shl EAX,4 ; EAX = линейный адрес PM_CODE
add EAX,offset ENTRY_POINT ; EAX = линейный адрес ENTRY_POINT
mov dword ptr ENTRY_OFF,EAX ; сохраняем его в переменной
; (кстати, подобный "трюк" называется SMC или Self Modyfing Code - самомодифицирующийся код)
```

Мы уже почти готовы переключится в защ. режим, единственное, что мы обязаны еще сделать – запретить все прерывания (причем, как маскируемые, так и немаскируемые), потому что пока мы не умеем с ними работать, и у нас нет ни одного обработчика, первый же тик таймера после переключения в PM завесит нам всю систему...

```
; запрещаем сначала маскируемые прерывания:
cli
; а затем и немаскируемые:
in AL,70h ; читаем 70h порт
or AL,80h ; ставим восьмой бит
out 70h,AL ; запихиваем на место
; теперь все прерывания запрещены
```

Для переключения в PM нужно установить нулевой бит регистра CR0:

```
mov EAX,CR0 ; EAX = CR0
or AL,1 ; ставим нулевой бит
mov CR0,EAX ; пихаем назад...
; загрузим в CS селектор на подготовленный сегмент кода. Мы не можем просто
взять и написать mov CS,селектор:
dd 66h ; префикс изменения разрядности операнда
db 0EAh ; опкод команды jmp far
dd ? ; смещение в сегменте, на которое мы jmp-аем
dw 00001000b ; селектор сегмента, в который мы jmp-аем
```

Разберемся, куда же мы все-таки прыгнули. Селектор равен 00001000b. Младшие два бита – нули, это уровень привилегий, на него не смотрим. Второй бит (TI) – нолик. Значит этот селектор указывает на дескриптор из таблицы GDT... На какой же? Смотрим левее... 001! На первый! А что у нас описывает первый дескриптор в GDT? Правильно, сегмент кода.

; значит, мы перелетели на некую «точку входа в защищенный режим», смещение которой мы уже высчитали

```
ENTRY_POINT:
; загрузим сегментные регистры селекторами на соответствующие дескрипторы:
mov AX,00010000b ; AX = селектор дескриптора данных (№2)
mov DS,AX ; кладем его в DS
mov AX,00011000b ; AX = селектор дескриптора видеопамяти (№3)
mov ES,AX ; кладем его в ES
xor SI,SI ; обнуляем SI
mov SI,PM_DATA ; SI = номер сегмента PM_DATA
shl ESI,4 ; ESI = линейный адрес сегмента PM_DATA
add ESI,offset message ; ESI = линейный адрес строки message
xor EDI,EDI ; EDI = позиция на экране (относительно 0B8000h)
mov ECX,mes_len ; длина текста в ECX
; вывод на экран:
rep movsb ; DS:ESI (наше сообщение) -> ES:EDI (видеопамять)
jmp $ ; погружаемся в вечный цикл
```

Теперь единственное, что мы можем сделать – это перезагрузить компьютер (причем, только кнопкой RESET).

Да, еще что. В полном исходнике сразу может напугать «Линия A20». Что это такое? Дело в том, что после запуска компа для совместимости с 8086 используются 20-разрядные адреса (адресные линии A0-A19), т.ч. попытка записать что-то по линейному адресу 100000h приведет к записи по адресу 0h. Вот этот режим нам и нужно отменить (установив бит 2 в 92h порту) для использования 32-х разрядной адресации.

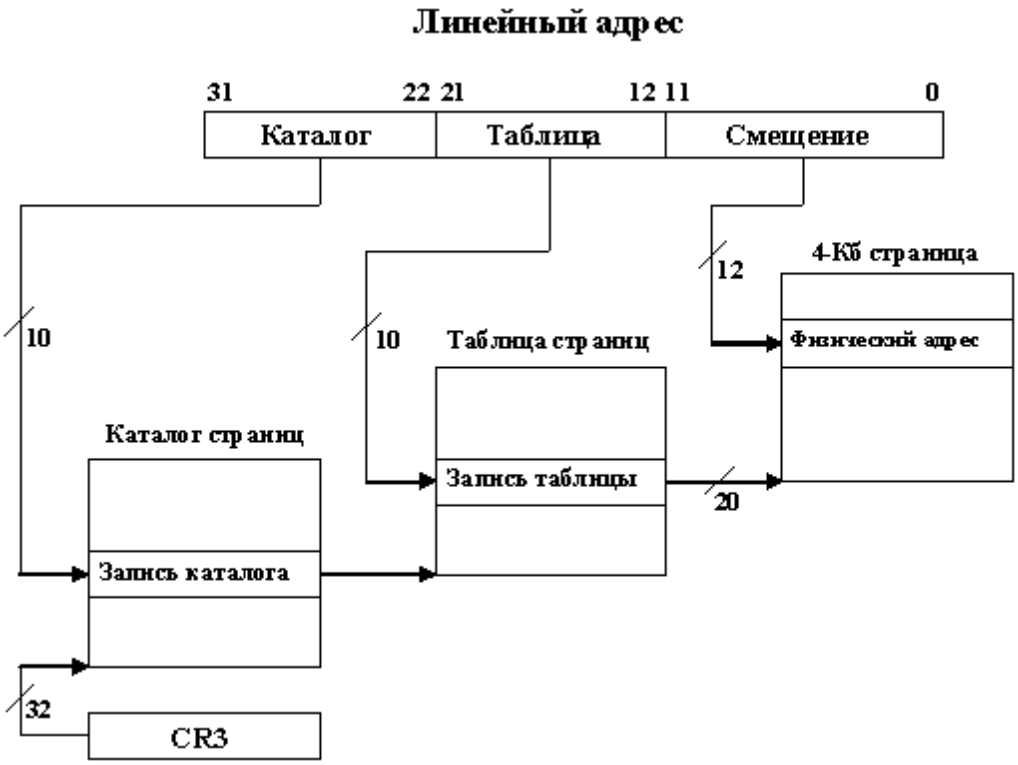
```
; открываем линию A20 (для 32-х битной адресации):
in AL,92h
or AL,2
out 92h,AL
```

А не могли бы мы, допустим, сделать базу видеосегмента в нуле, а при выводе на экран значение 0B8000h поместить в EDI? Конечно могли бы, это то же самое по сути.

Использование страничной адресации

Линейный адрес, который формируется процессором из логического адреса, соответствует адресу из линейного непрерывного пространства памяти. В обычном режиме в это пространство могут попадать области памяти, куда нежелательно разрешать запись, - системные таблицы и процедуры, ПЗУ BIOS и т.д. Чтобы этого избежать, система может позволять программам создавать только небольшие сегменты, но тогда теряется привлекательная идея flat-памяти. Сегментация - не единственный вариант организации

памяти, который поддерживают процессоры Intel. Существует второй, совершенно независимый механизм - страничная адресация (paging).
При страничной адресации непрерывное пространство линейных адресов памяти разбивается на страницы фиксированного размера (обычно 4 Кб (4096 или 1000h байт), но Pentium Pro может поддерживать и страницы по 4 Мб). При обращении к памяти процессор физически обращается не по линейному адресу, а по тому физическому адресу, с которого начинается данная страница. Описание каждой страницы из линейного адресного пространства, включающее в себя ее физический адрес и дополнительные атрибуты, хранится в одной из специальных системных таблиц, как и в случае сегментации, но при этом страничная адресация абсолютно невидима для программы.
Страничная адресация включается при установке бита PG регистра CRO, если бит PE зафиксирован в 1 (попытка установить PG, оставаясь в реальном режиме, приводит к исключению #GP). Кроме того, в регистр CR3 предварительно надо поместить физический адрес начала каталога страниц - главной из таблиц, описывающих страничную адресацию. Каталог страниц имеет размер 4096 байт (ровно одна страница) и содержит 1024 4-байтных указателя на таблицы страниц. Каждая таблица страниц тоже имеет размер 4096 байт и содержит указатели до 1024 4-килобайтных страниц. Если одна страница описывает 4 Кб, то полностью заполненная таблица страниц описывает 4 Мб, а полный каталог полностью заполненных таблиц - 4 Гб, то есть все 32-битное линейное адресное пространство.
Когда процессор выполняет обращение к линейному адресу, он сначала использует его биты 31—22 как номер таблицы страниц в каталоге, затем биты 21—12 как номер страницы в выбранной таблице, а затем биты 11-0 как смещение от физического адреса начала страницы в памяти. Поскольку эта процедура занимает много времени, в процессоре предусмотрен специальный кэш страниц - TLB (буфер с ассоциативной выборкой), так что, если к странице обращались не очень давно, ее физический адрес будет сразу определен.
Второе не менее важное применение страничной адресации - безопасная реализация flat-модели памяти. Операционная система может разрешить программам обращаться к любому линейному адресу, но отображение линейного пространства на физическое не будет взаимно однозначным. Скажем, если система использует первые 4 Кб памяти, физическим адресом нулевой страницы будет не ноль, а 4096 и пользовательская программа даже не узнает, что обращается не к нулевому адресу.
По сути - код остался прежним, я лишь опишу изменения, которые необходимо внести.
Следует иметь ввиду, что все изменения для включения страничной адресации производятся уже после перехода в защищенный режим. И еще - страницы у нас будут 4-килобайтными.
И вот они, эти изменения:
Подготовить каталог страниц
Заполнить таблицу страниц
Адрес каталога страниц поместить в регистр CR3
Включить страничную адресацию (установка бита 31 в регистре CR0)
Вот картинка (она уже у нас фигурировала), из которой видно для чего нужны эти 4 пункта:



Прежде, чем приступить, хочу вкратце рассказать, чего же мы будем творить. Вопрос в том, как наглядно показать разницу между сегментной и сегментно-страничной адресацией?
Мы рассмотрели код, который выводит строку на экран, т.е. все свелось в конце концов к тому, что по адресу ES:0 вывелась надпись, где ES - селектор на предварительно подготовленный дескриптор, база которого равна 0B8000h (т.е. начало видеопамяти в текстовом режиме). Все однозначно и понятно. При страничной адресации адрес 0B8000h препарируется на три части: старшие 10 бит - номер элемента в каталоге страниц, средние 10 бит - номер элемента в таблице страниц, и, наконец, младшие 12 бит - это смещение в странице.
Недолго думая, можно заключить, что, попытавшись что либо записать по адресу 0B8000h при страничной адресации, мы не обнаружим там никакого начала видеопамяти, разумеется, если предварительно мы корректно не настроим каталог и таблицы страниц. Вроде, вот же он: адрес начала видеопамяти, 0B8000h – но нет, этот адрес - не линейный, а виртуальный. Вот разбить его на три части, пройтись по каталогу и таблице страниц - вот тогда сформируется окончательный адрес (выставляемый процем на шину).
Поэтому вот оно, золотое правило: при страничной адресации линейный адрес формируют элементы каталогов и таблиц страниц, ну и конечно "проводником" по этим таблицам служит сам виртуальный адрес...
Если представить все это дело наоборот, то придем точно к такому же выводу. Т.е. если 0B8000h - это на самом деле нечто совершенно иное, то с точно таким же успехом, любой виртуальный адрес после преобразований может превратиться в 0B8000, нам стоит только захотеть этого. А что значит захотеть? А это значит, соответствующим образом заполнить элементы каталога и таблицы страниц, только и всего...
Вот что мы еще сделаем: вообще уберем дескриптор сегмента видеопамяти из таблицы GDT (итого у нас останется два дескриптора - кода и данных). Для чего - станет ясно ниже.
Итак, где-нибудь сразу после перехода в защ. режим и загрузки сегментных регистров соотв. селекторами нужно создать каталог страниц.

```
ENTRY_POINT:
; загрузим сегментные регистры селекторами на соответствующие дескрипторы:
mov AX,00010000b ; селектор на второй дескриптор (DATA_descr)
mov DS,AX ; в DS его
mov ES,AX ; и в ES его же
```

вот теперь создадим каталог страниц. Что это такое? Каталог страниц - это набор 32- разрядных записей (элементов).
Начало каталога страниц в оперативной памяти будет располагаться по адресу 1Мб (100000h).
В нашем примере единственное, что мы делаем полезного - выводим на экран некоторую надпись. По сути, при этом используется один единственный адрес - ES:ESI, или 0B8000h. Следовательно, в данном

примере для демонстрации возможностей страничной адресации достаточно заполнить лишь один единственный элемент каталога страниц. Так и сделаем:

```
mov EDI,100000h ; начало каталога страниц (1Мб)
mov EAX,101007h ; это наш один единственный значащий элемент
stosd ; ES:[EDI] <- EAX
mov ECX,1023 ; остальные 1023 элементов
xor EAX,EAX
rep stosd ; забьем нулями все остальные элементы каталога
```

кстати, почему такой загадочный элемент - 101007h? 7 (установлены младшие 3 бита адреса) - означает, что страница присутствует в оперативной памяти (бит P), доступна для чтения записи (бит R/W) и доступна с любого уровня привилегий (бит U/S). А что такое 1010? Не забыл, что младшие 12 бит для адреса таблицы страниц всегда равны нулю? Значит, это 1010000h, что соответствует 1Мб + 4Кб, а 4 Кб - потому что сам каталог занимает столько.

С каталогом покончено. Теперь примемся за таблицу страниц

```
mov EAX,00000007h ; первая запись - адрес нулевой страницы равен 0
mov ECX,1024 ; кол-во страниц в таблице
fill_page_table:
stosd ; запишем первый элемент
add EAX,1000h ; добавим 4 Кб
loop fill_page_table ; и повторим для всех элементов таблицы страниц
mov EAX,00100000h ; базовый адрес = 1 Мб
mov CR3,EAX ; в CR3 его! (база каталога страниц ВСЕГДА должна лежать в CR3)
; включить страничную адресацию
mov EAX,CR0
or EAX,80000000h
mov CR0,EAX
; а теперь изменить физический адрес страницы 12000h на 0B8000h
mov EAX,000B8007h
mov ES:00101000h+012h*4,EAX
```

вот здесь надо проникнуться всем существом. Мы, в таблице страниц, заменили адрес страницы, начало которой равно 12000h байт на адрес начала видеопамяти (0B8000h). Адрес начала таблицы страниц = 101000h, так? А сколько занимает один элемент? 4 байта. (32 бита). Поэтому 12*4 - это и есть запись таблицы страниц, которая указывает на страницу, начинающуюся по адресу 12000h.

А вот теперь демонстрация силы и мощи страничной адресации:

```
; вывод mes1 по стандартному адресу (начало видеопамяти 0B8000h)
mov EDI,0B8000h ; для команды movsw, EDI = начало видеопамяти
mov ESI,PM_DATA
shl ESI,4
add ESI,offset mes1 ; ESI = адрес начала mes1
mov ECX,mes_len ; длина текста в ECX
rep movsw ; DS:ESI (наше сообщение) -> ES:EDI (видеопамять)
; вывод mes2 по НЕСТАНДАРТНОМУ АДРЕСУ 12000h:
mov EDI,0120A0h ; 12000h (уже можешь считать, что это 0B8000h) + A0h
mov ESI,PM_DATA
shl ESI,4
add ESI,offset mes2 ; ESI = адрес начала mes2
mov ECX,mes_len ; длина текста в ECX
rep movsw ; DS:ESI (наше сообщение) -> ES:EDI (видеопамять)
```