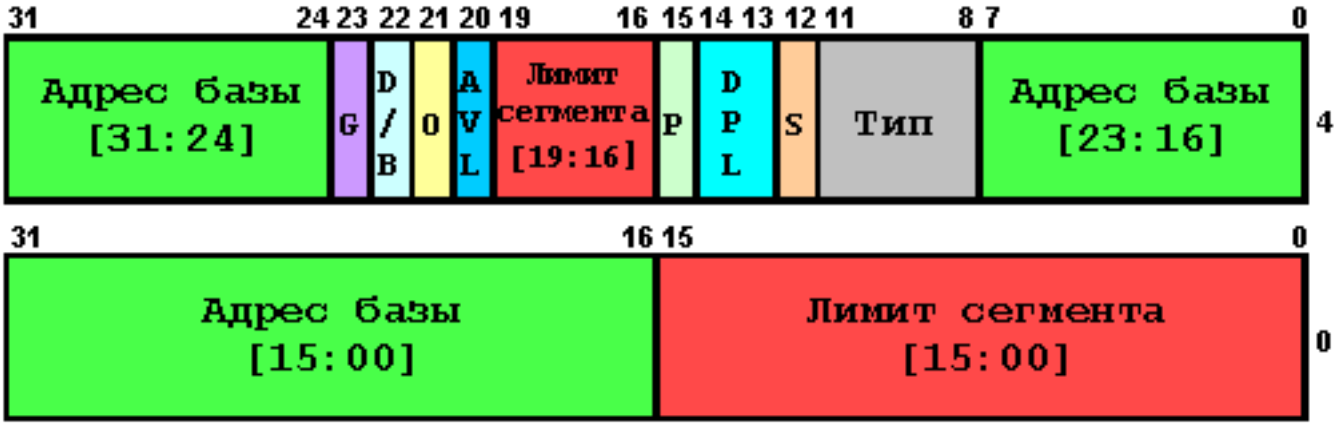
Речь в грядущем цикле лекций пойдет о программировании в защищенном режиме процессоров Intel, причем даже не столько о программировании, сколько о самой архитектуре, о сущности защищенного режима.
Ясное дело, защищенный режим и программирование на высоких уровнях – совершенно разные вещи, как и всё то, что мы уже успели изучить на лекциях, не имеет чего-то особо общего с программированием под операционными системами типа windows – есть принципиальная разница между режимами работы процессора и операционными системами.
Реальный режим остался в прошлом веке, в процессорах до 8086. Но и архитектура защищенного в том виде, который мы будем изучать на наших лекциях, присутствовала в процессорах Intel вплоть до Pentium 4. Основа этого режима, несмотря на это, была заложена уже тогда, и в современных она лишь модифицируется, подстраиваясь под обновленное железо.
Главная проблема – с чего начать изучение режима. Управление защищенным режимом в процессорах Intel – один из самых сложных частей в программировании под него, но ничего невозможного нет. Также проблема в том, что в изучении защищенного режима никак не обойтись без сухих и нудных описаний полей дескрипторов, структур системных сегментов. Тем не менее, я постараюсь объяснять так, чтобы это не было похоже на скучное руководство пользователя, и здесь же помогут познания реального режима, который мы, по сути, изучали на прошлых лекциях.
Итак. Что вообще такое защищенный режим? Дело в том, что реальный режим процессора Intel – однозадачная среда, в данный момент времени в ней может выполняться только одна конкретная задача. Безусловно, можно сэмулировать многозадачность и в реальном режиме, но все дело в том, что именно в защищенном режиме вся многозадачность реализована аппаратно, и, по сути, это и является единственным принципиальным отличием между режимами. Конечно, это отличие достигается сложными структурами вроде дескрипторов, селекторов и разных системных структур, но об этом позже.
Для начала рассмотрим общие положения об организации памяти в РМ. В процессорах Intel организацию памяти разделяют на две части: сегментация (segmentation), и страничная организация (paging).
Сегментация позволяет изолировать модули кода, данных, стека и позволяет работать нескольким задачам и программам на одном процессоре (как говорится, multitasking) и не конфликтовать между собой – это и есть по большому счету, вся революция по сравнению с 8086. Страничная организация памяти, вообще говоря, сложнее и более громоздкая, чем сегментация.
В процессоре нет такого бита, который бы четко отвечал за переключение между этими двумя режимами. Элементы сегментной организации, в любом случае, присутствуют всегда. А вот уже использовать страничную организацию или нет – выбор за нами (за это отвечает флаг PG, бит 31 регистра CR0). CR0 – это обычный регистр (такой же, как AX, BP и т.д.), подробнее поговорим о нем чуть позже.
Сегментация – это механизма разделения адресного пространства процессора (по-русски: память, которую «видит» процессор) на отдельные защищенные друг от друга кусочки (сегменты). Естественно, сразу возникает вопрос, а от чего защищать? По мере лекций он отпадёт сам собой.
Непосредственно сегмент может содержать в себе код, данные, стек, системные структуры данных. Если запущено сразу несколько программ, то каждой программе принадлежит своя, личная группа сегментов.
Итак, что мы имеем. Память, разделена на группы сегментов, у каждой группы есть свой владелец (программа). Программы плотно расположены по адресам оперативной памяти и чужие программы к своим сегментам «не подпускают». При любой попытке пресечь границы возникает эдакая тревога, вследствие чего вдруг откуда ни возьмись возникает исключение #GP (General Protection), а все дальнейшие действия уже осуществляются засчёт того, как поведет себя обработчик этого исключения.
Обработка исключительных ситуаций (англ. exception handling) — механизм языков программирования, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (исключения), которые могут возникнуть при выполнении программы и приводят к невозможности (бессмысленности) дальнейшей отработки программой её базового алгоритма.
Для того, чтобы обратиться к любому байту в любом сегменте в памяти мы должны сформировать логический адрес (или дальний указатель).

Логический адрес представляет собой селектор и смещение. Селектор – это уникальный идентификатор сегмента, у каждого сегмента есть свой селектор. Селектор при обращении к памяти содержится в сегментном регистре (те самые DS, CS, ES …). Зная селектор и смещение, мы можем получить линейный адрес – место, где реально расположен нужный нам байт. Как именно его получить – узнаем позже, поскольку для этого нужно добавить ещё одно важное определение - дескриптор (структура, описывающая сегмент).
Существует также такое понятие, как физическое адресное пространство процессора. Физический адрес – это адрес, который процессор может выставить на адресную шину, и в случае сегментной организации памяти (но не страничной) совпадает с линейным.
Виды памяти в защищенном режиме
Простая плоская модель
Действительно, самая простая модель: вся память представляет одно огромное адресное пространство, никакого механизма распределения, никаких сегментов, ничего нет. Тем не менее, должно быть минимум два дескриптора. Один из этих дескрипторов должен описывать сегмент кода (с началом в 0 и лимитом в 4 Гб), второй – сегмент данных (также с началом в 0 и лимитом в 4 Гб). Они «накладываются друг на друга», в этой модели памяти нет никакой защиты.
Защищенная плоская модель
Все отличие от предыдущей модели – база и лимит кода и данных уже не совпадают, и здесь уже исключение #GP может себя проявить. Более того, если включить флаг PG (страничная адресация), то мы получим защищенную плоскую модель с страничной адресацией. Но здесь нам уже понадобится минимум 4 сегмента: для кода и данных на уровне привилегий 3 и для кода и данных на уровне 0 (процессор часто использовал только два этих уровня привилегий, оставляя 3 для пользовательских прог, а 0 – для ядра).
Мульти-сегментная модель
Данная модель использует все возможности процессора на полную, позволяет аппаратно защищать код, структуры данных, задачи и программы друг от друга.

Начнем с подробного рассказа о сегментной модели.
Итак, что мы имеем: программа (вернее, программы) загружены в оперативную память, каждая программа владеет группой сегментов (данные, код, стек). Теперь возникает вопрос: откуда же процессор знает, где какой сегмент? Где начало сегмента? И где его конец? И что это за сегмент: данных, кода, стека? Ответить на этот вопрос может только одна структура данных: дескриптор. Дескриптор – это структура, описывающая сегмент, у каждого сегмента он собственный.



Естественно, требуется понимать значения всех его полей, без этого понимания того, как вообще работает защищенный сегментный режим, не будет.
Адрес базы: адрес нулевого байта описываемого сегмента в 4 Гб линейном адресном пространстве (т.е. адрес, с которого начинается сегмент). Процессор собирает в кучу три поля цвета и образует единый 32-х битный адрес.
Лимит сегмента: определяет размер сегмента. Опять же, процессор собирает в кучу два поля и формирует конечный, 20-битовый размер. Реальный лимит сегмента зависит от бита гранулярности (G-granularity);

- если бит гранулярности сброшен (0), то 20-битное значение и будет тем самым лимитом сегмента
- если бит гранулярности установлен (1), то всё 20-битное значение автоматически увеличивается в 1000h раз, т.е. если при G=0 мы измеряем размер в байтах, то при G=1 – в 4Кб единицах.

Например, если G=1 и поле «Лимит сегмента» = 0000Fh (15 байт), то реальный лимит (читай – размер) данного сегмента равен 0Fh*1000h=0F000h (61440). Но: к лимиту прибавляется либо 1, либо 1000h в зависимости от бита G. Сделано это вот для чего:

Следует отметить, что если поле «Лимит сегмента» содержит значение равное 0, то это значит, что описанный сегмент имеет размер в 1 байт (а не ноль!) при G=0, и размер в 4Кб при G=1. Т.е. сегмент никак не может иметь нулевую длину, минимум – 1 байт, максимум – 0xFFFFFh * 4Кб = 4 Гб.

Есть еще один бит, от которого зависит смысловое значение этого поля. Бит направления роста сегмента (B-big));

- если этот бит сброшен (0), то разрешены все смещения от 0 до лимита
- если установлен (1) – то все, кроме от 0 до лимита.

В противном случае (если мы попытаемся обратится за пределы лимита) – возникнет исключение главной защиты (#GP, general protection). Вообще говоря, лимиты для этой цели и придуманы – отслеживать обращения в недоступные адресные пространства.

Лимит, очевидно, отсчитывается от адреса базы.

Тип:

Определяет тип сегмента, определяет права доступа к сегменту и направление роста сегмента (помните бит B). Значение этого поля зависит от значения поля «тип дескриптора» (S-descriptor type). Значение этого поля различно для разных типов сегментов (кода, данных и системного)

S (descriptor type) – флаг «тип дескриптора»:

Означает только одно: если сброшен (0), то описуемый сегмент – системный, если установлен (1) – это сегмент данных или кода.

DPL (descriptor privilege level) – уровень привилегий дескриптора:

Определяет уровень привилегий сегмента. Т.к. это поле – двухразрядное, то соответственно может принимать только четыре различных значения (от 0 до 3). Самый крутой – нулевой уровень привилегий (ядро ОС). Это поле нужно для контроля за доступом к сегменту.

P (segment present flag) – флаг присутствия сегмента:

Если установлен, значит сегмент присутствует непосредственно в памяти; если сброшен – соответственно, отсутствует. Этот флаг предназначен для организации работы при использовании страничной адресации (дело в том, что когда в сегментный регистр грузят селектор на дескриптор, в котором сброшен этот бит – возникает исключение #NP (segment-not- present exeption)), поэтому, если отлавливать это самое #NP, можно вовремя подгрузить новую страницу в ОП из файла подкачки.

G (Granularity) – флаг «гранулярности»:

См. «Лимит сегмента». Еще раз повторю - этот флаг влияет на лимит сегмента: если сброшен – лимит измеряется в байтах, если установлен – в 4 Кб единицах. На адрес базы этот флаг не влияет, только на лимит – база всегда измеряется в байтах.

AVL (Available and reserved bits) – зарезервировано:

Это два бита (21-20 во втором двойном слове). Они вообще не стоят того, чтобы на них заострять внимание, но все же может кому и пригодиться: бит 20 может использоваться как угодно по вашему усмотрению, бит 21 всегда должен быть равен нулю.

Остался последний бит, 22. D/B. Зависит от типа сегмента (в зависимости от этого он называется либо D, либо B). Учитывая то, что про него уже было сказано при описании лимита сегмента, следует отметить еще вот что: он определяет разрядность сегмента. Если установлен – значит сегмент 32-х разрядный, если сброшен – 16-ти. Это общий случай. Теперь частные:

- Сегмент кода. Для сегмента кода данный флаг называется D и устанавливает длину по умолчанию для эффективных адресов и операндов в сегменте. Если установлен – то в сегменте допустимы 32-х битные адреса и 32-х битные ИЛИ 8-ми битные операнды; Если сброшен – 16-битные адреса и 16-битные ИЛИ 8-ми битные операнды.
- Сегмент стека. Для сегмента стека данный флаг называется B (big) флаг и устанавливает длину указателя стека (регистр ESP) для команд push, pop и call. Если установлен – используется 32-х разрядный указатель стека (т.е. регистр ESP использован по максимуму), если сброшен – то используется только 16 бит (регистр SP, уже без буковки E).

Итак, к чему мы пришли: сегменты не раскиданы по памяти как попало и непонятно где, теперь мы с уверенностью можем сказать где какой из сегментов начинается, где заканчивается, что это за сегмент (код, данные или стек), вся эта информация хранится в дескрипторе. Но где находится сам дескриптор? Находиться он может в трех местах:
Глобальная таблица дескрипторов (GDT - Global Descriptor Table)
Локальная таблица дескрипторов (LDT - Local Descriptor Table)
Таблица дескрипторов прерываний (IDT – Interrupt Descriptor Table)
Эти таблицы находятся в оперативной памяти (там же, где и собственно сами программы). Поэтому, очевидно, что строить их придется «руками», нам самим. В защищенном режиме все в целом нужно строить самому. Естественно, это открывает больше возможностей ошибок, но в этом же и кроется «сила».
Несколько слов по поводу первых двух таблиц (на третьей (IDT) мы остановимся в главе посвященной прерываниям в защищенном режиме).
Глобальная таблица дескрипторов (GDT):
Каждая ОС должна иметь одну таблицу GDT. Ей могут пользоваться все программы и задачи системы. Что значит «пользоваться таблицей дескрипторов»? Это значит, хранить в ней свой дескриптор.
Таблица GDT – сама по себе не сегмент. Это структура данных в линейном адресном пространстве (в памяти). Начало таблицы GDT храниться в регистре GDTR. Регистр GDTR – это самый обыкновенный регистр, такой же обыкновенный, как EAX, EIP, ES, только вот его функция заключается не в хранении каких-то промежуточных данных, а в хранении фиксированного числа – начала таблицы GDT. Называние регистра GDTR запомнить очень легко: GDT – это таблица, R – регистр, в итоге получаем global descriptor table register.
Адрес начала таблицы GDT в памяти должен быть кратен 8. Это связано с архитектурой, так процессор быстрее работает при обращении к таблице.
GDTR:
32-битный линейный базовый адрес|16-битный лимит таблицы
Вот он, регистр GDTR. Весит 48 бит. Он содержит не только адрес начала таблицы GDT в памяти, а еще и ее лимит. Лимит таблицы – 16-битное значение, показывает величину таблицы в байтах + 1. (т.е. все как и в случае с лимитом сегмента: если лимит таблицы в GDTR равен 0, то на самом деле это означает что реально (в памяти) лимит равен одному байту).
Сегментный дескриптор всегда занимает 8 байт (2 двойных слова). Следовательно, лимит таблицы дескрипторов – величина, равная 8N-1 байт.
Первый дескриптор в GDT не используется и называется «нулевой дескриптор» (null descriptor). При обращении к памяти через этот дескриптор возникает уже знакомое исключение #GP, general protection. Поэтому первый дескриптор в GDT трогать нельзя.
Загрузить/считать значение регистра GDTR можно командами LGDT/SGDT. По умолчанию (т.е. после нажатия на кнопку Reset или включения компьютера) база GDT равна нулю, а лимит – FFFFh, т.е. фактически по умолчанию выделено максимум места, под FFFFh/8 = 8191 дескрипторов (минус один, учитывая null descriptor).
Локальная таблица дескрипторов (LDT):
В отличии от GDT совершенно не обязана присутствовать вообще. И в то же самое время, по желанию, их можно развести великое множество (GDT должна быть только одна). Каждая задача может иметь свою собственную LDT, в то же время несколько задач могут использовать одну LDT на всех.
LDT – это сегмент (GDT – структура данных). Это принципиально, потому что так как LDT – это сегмент, то значит у нее тоже есть свой дескриптор в той же глобальной таблице дескрипторов.
Так же, как и у GDT, у LDT тоже есть свой регистр – LDTR. В отличии от GDTR этот регистр, помимо инфы про базу и лимит LDT, содержит еще одно поле – сегментный селектор.
LDTR:
Сегм. Селектор (16 бит)|32-битный линейный базовый адрес|16-битный лимит сегмента
Инструкции LLDT и SLDT позволяют писать/читать регистр LDTR. Точно так же, при reset-е значение базы в LDTR падает в ноль, а лимит – в FFFFh.
Пример GDT.
Представляю пример структуры, которая являет собой таблицу GDT. Поработаем над ними и найдем для них реальные значения базы и лимита.

```
GDT:
; нулевой дескриптор (ОБЯЗАТЕЛЬНО ДОЛЖЕН ПРИСУТСВОВАТЬ, НО РУКАМИ НЕ ТРОГАТЬ!)
0.                      db    8 dup (0)

; Сегмент с  базой в 0 и лимитом в 1235h (не забывай про +1 к лимиту!)
; Линейный адрес базы = 0, линейный адрес лимита = 0 + 1235h = 1235h
1. Descr_code           db    34h,12h,00h,00h,00h,XXh,0X000000b,00h

2. Descr_data           db    0C8h,0Dh,36h,12h,00h,XXh,0X100000b,00h
3. Descr_stack          db    0FFh,00h,00h,20h,00h,XXh,1X000000b,00h
4. Descr_code2          db    0DEh,0BCh,01h,20h,10h,XXh,0X001010b,00h
5. Descr_data2          db    00h,00h,00h,00h,00h,XXh,0X000000b,10h
6. Descr_stack2         db    01h,00h,10h,00h,00h,XXh,0X000001b,10h
7. Descr_LDT            db    04h,00h,00h,00h,00h,XXh,1X000000b,20h

GDT_limit                =    $-GDT

GDTR                    dw    GDT_limit-1
                        dd    ?
```

1. сегмент с базой = 0 и размером = 1235h

2. сегмент с базой = 1236h и размером = 0DC9h

3. сегмент с базой = 2000h и размером = 100000h

4. сегмент с базой = 102001h и размером = 0ABCDFh

5. сегмент с базой = 10000000h и размером = 1

6. сегмент с базой = 10000010h и размером = 10002h

7. сегмент с базой = 20000000h и размером = 5000h

Ошибка во втором дескрипторе: бит 21 во втором двойном слове дескриптора ДОЛЖЕН ВСЕГДА равняться нулю.

Далее справочная информация о поле «типа дескриптора». Все, что сейчас будет рассказано, справедливо только при S=1 (то есть сегмент либо кода, либо данных).
11 бит во втором двойном слове (он же четвертый бит поля «Тип») показывает, является описуемый сегмент кодом или данными (0 – данные, 1 – код)
Если это сегмент данных, то младшие три бита поля «Тип» (10-8) интерпретируются как бит E (expansion-direction) – рост направления, бит W (write-enable) – запись разрешена и бит A (accessed) – доступен.

№	Поле Тип				Тип дескриптора	Описание
	11	10 E	9 W	8 A		
0	0	0	0	0	Данные	Только для чтения
1	0	0	0	1	Данные	Только для чтения, доступен
2	0	0	1	0	Данные	Для чтения/записи
3	0	0	1	1	Данные	Для чтения/записи, доступен
4	0	1	0	0	Данные	Только для чтения, растет вниз
5	0	1	0	1	Данные	Только для чтения, растет вниз, доступен
6	0	1	1	0	Данные	Для чтения/записи, растет вниз
7	0	1	1	1	Данные	Для чтения/записи, растет вниз, доступен
		C	R	A		
8	1	0	0	0	Код	Только для исполнения
9	1	0	0	1	Код	Только для исполнения, доступен
10	1	0	1	0	Код	Для исполнения/чтения
11	1	0	1	1	Код	Для исполнения/чтения, доступен
12	1	1	0	0	Код	Только для исполнения, подчинен
13	1	1	0	1	Код	Только для исполнения, подчинен, доступен
14	1	1	1	0	Код	Для исполнения/чтения, подчинен
15	1	1	1	1	Код	Для исполнения/чтения, подчинен, доступен

И вот здесь вас подстерегает один приятный сюрприз: оказывается, нет такого специального типа сегмента, как сегмент стека. Сегмент стека – это ни что иное, как сегмент данных, растущий вниз. Сегмент стека обязательно должен быть доступен для чтения/записи.
Для сегмента кода младшие три бита поля «Тип» интерпретируются, как C (conforming) – подчиненный, R (read enable) – чтение разрешено и A (accessed) – доступен. Как видите, сегмент кода может быть доступен как только для исполнения, так и для исполнения/чтения. Для исполнения/чтения он может быть доступен в том случае, когда мы храним какие-либо константы или другие статические данные в памяти, непосредственно в сегменте кода. В защищенном режиме в сегмент кода писать нельзя.

Теперь, если S=0 (то есть сегмент системный):
Процессор различает 6 типов системных дескрипторов:

Дескриптор сегмента «таблица LDT»
Дескриптор сегмента «состояние задачи»
Дескриптор шлюза вызова (Call-Gate)
Дескриптор шлюза прерывания (Interrupt-Gate)
Дескриптор шлюза ловушки (Trap-Gate)
Дескриптор шлюза задачи (Task-Gate)
В свою очередь, данные 6 дескрипторов делятся на две группы: дескрипторы системных сегментов (первые два) и дескрипторы шлюзов (все остальные).
Самый первый (дескриптор сегмента «таблица LDT») нам уже хорошо знаком, остальные являются более сложными. Тем не менее, таблица их распределения:

№	Поле «Тип»				Описание
	11	10	9	8	
0	0	0	0	0	Зарезервировано
1	0	0	0	1	16-битный TSS (свободен)
2	0	0	1	0	LDT
3	0	0	1	1	16-битный TSS (занят)
4	0	1	0	0	16-битный шлюз вызова
5	0	1	0	1	Шлюз задачи
6	0	1	1	0	16-битный шлюз прерывания
7	0	1	1	1	16-битный шлюз ловушки
8	1	0	0	0	Зарезервировано
9	1	0	0	1	32-битный TSS (свободен)
10	1	0	1	0	Зарезервировано
11	1	0	1	1	32-битный TSS (занят)
12	1	1	0	0	32-битный шлюз вызова
13	1	1	0	1	Зарезервировано
14	1	1	1	0	32-битный шлюз прерывания
15	1	1	1	1	32-битный шлюз ловушки

Итак, сначала мы узнали, что программа состоит из сегментов и все они расположены в памяти. Каждый сегмент описывает специальная структура – дескриптор. Дескриптор хранится в специальной таблице. Найти в памяти таблицу можно по специальному регистру (GDTR, LDTR). Двигаемся дальше.

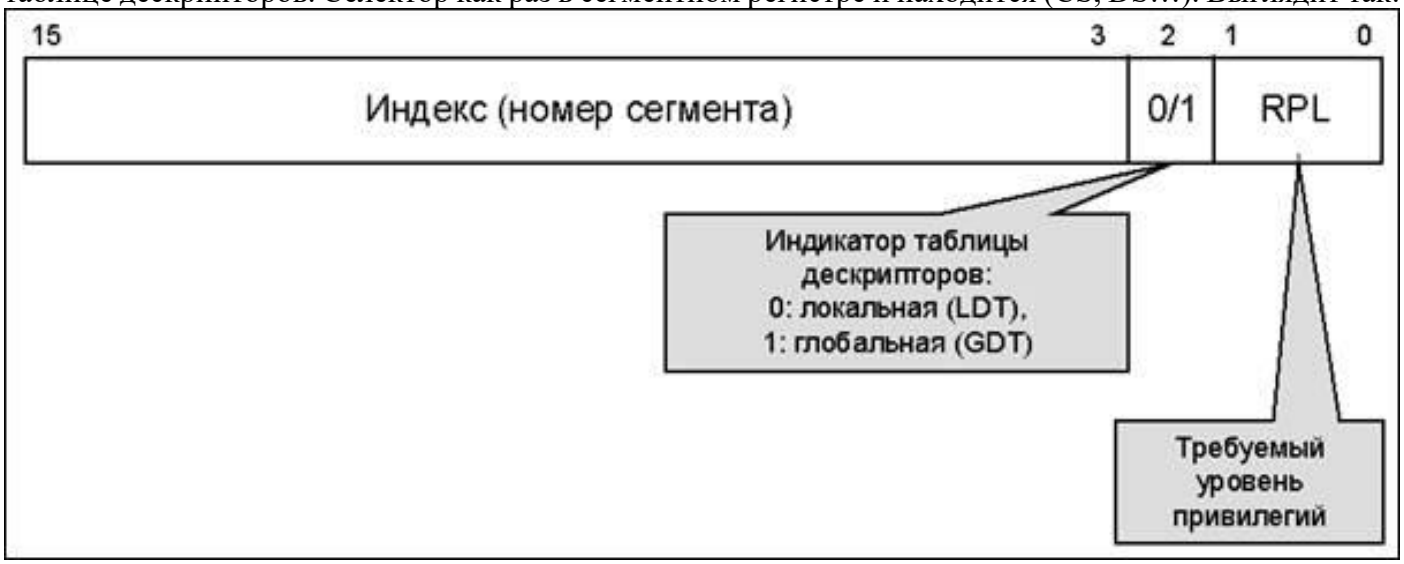
Селектор

Пока у нас никак не были задействованы сегментные регистры.

Мы все это время спускаемся вниз по ступенькам:

сегмент в памяти <---- дескриптор <---- таблица дескрипторов …

Следующей ступенью будет селектор. Это 16-битная структура данных, которая является идентификатором сегмента. Селектор указывает не на сам сегмент в памяти, а на его дескриптор, в таблице дескрипторов. Селектор как раз в сегментном регистре и находится (CS, DS…). Выглядит так:



Поле индекс (биты 3-15): указывает на один из 8192 дескрипторов в таблице дескрипторов (GDT или LDT). Почему 8192? Это наибольшее число, которое поместится в 13 битах. (2^13).

Процессор сам умножает значение поля индекс на 8 и добавляет к полученному значению адрес базы таблицы. То есть процессор умножит то, что находится в поле «индекс», на 8, а потом прибавит значение регистра таблицы – и мы благополучно указываем на начало дескриптора.

- Как же узнать, из какой именно таблицы дескриптор?

Для этого нужен флаг TI (table indicator) (второй бит). Если он = 0, то прибавится значение регистра GDTR (т.е. дескриптор расположен в таблице GDT), если же установлен – LDTR.

Допустим, мы кладем в DS число 0000000000110 0 00b. Что это значит? Сразу разбиваем DS на кусочки (15-3 биты – индекс, 2 – TI, 1-0 – приивлегии). Индекс равен 6. Значит, шестой по счету дескриптор. А где? В GDT (TI=0).

Кладем в ES число 0000000001000 1 00b. Восьмой дескриптор. На этот раз в LDT.

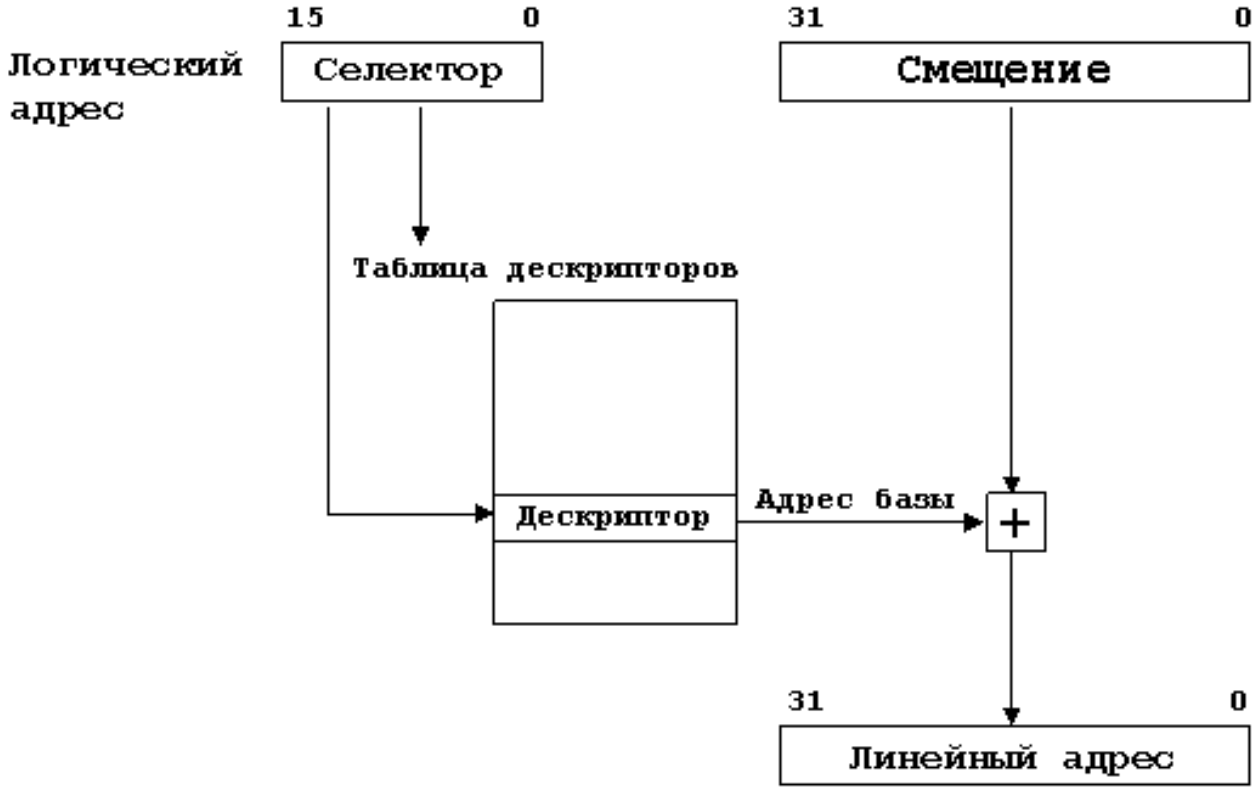
А учитывается ли нулевой дескриптор (null descriptor) при «счете»?

Обязательно. Более того, мы даже можем положить в сегментный регистр (DS, SS…) селектор с полями индекс и TI равными 0. То есть, фактически мы выбираем нулевой дескриптор в таблице GDT. Это возможно, и ничего страшного не произойдет до тех пор, пока мы не обратимся к памяти, используя такой сегментный регистр. А так он может пролежать там сколько угодно времени, но как только мы обратимся к памяти используя такой регистр (с индексом = 0) – мгновенное исключение #GP, general protection.

Преобразование логического адреса в линейный

И вот мы подобрались к самому важному моменту: как же все-таки процессор формирует адрес и знает, куда обращаться? Для этого нужно собрать все полученные нами знания в кучу. Что для этого нужно сделать? Допустим, нам нужно узнать адрес инструкции в памяти, но о ней не известно ничего, кроме того, что на нее указывает CS:EIP. По сути это логический (то есть некий абстрактный адрес). Как же найти линейный адрес в памяти, руководствуясь только этими двумя значениями: CS и EIP? Сразу смотрим в CS и ищем в нем поле "Индекс" (см. на селектор выше). Смотрим в поле индекс и тут же узнаем о местоположении нужного дескриптора в таблице дескрипторов. Далее нам нужно узнать адрес сегмента. Узнали. Теперь осталось только одно: сложить этот адрес базы с EIP - и мы получим

линейный адрес инструкции в памяти (который при сегментной организации совпадает с физическим, мы об этом говорили ранее). Еще раз: селектор-->дескриптор-->база... +EIP = линейный адрес.



Преобразование логического адреса в линейный

Сегментный регистр

В архитектуре процессоров Intel существуют ШЕСТЬ сегментных регистров:
CS, DS, SS, ES, GS и FS. Каждый из этих регистров отвечает за свой сегмент в памяти (кода, данных или стека). Итак, даже если программа состоит из тысячи сегментов, только 6 из них могут быть доступны в данный момент времени. Другие сегменты станут доступны только после загрузки соответствующих селекторов в сегментные регистры.

В реальном режиме линейные адреса формировались просто: значение сегментного регистра умножалось на 10h и прибавлялось смещение. То есть никакого селектора явно не существовало, никаких дескрипторов, только сегментный регистр и смещение. В защищенном же режиме нужно пройти несколько шагов (селектор-дескриптор- база...) чтобы докопаться до линейного адреса... Итак, в защищенном режиме сегментный регистр - это 16-битный регистр, содержащий информацию о дескрипторе (а именно - местоположении) и запрашиваемом уровне привилегий. Здесь сразу же возникает вопрос: неужели же процессор при каждом обращении к памяти все время повторяет одни и те же действия (ищет дескриптор, потом ищет базу, затем прибавляет к базе смещение...), и так при выполнении фактически каждой команды. На самом деле сегментный регистр - это 80-битный регистр, нам же доступны только младшие 16 бит, которые и называются селектором. Остальные 64 бита называются "Теневым регистром" (Shadow register) или "Дескрипторным кэшем" (Descriptor cache), они и содержит ту самую базу, которую процессор по идее должен был бы высчитывать на каждом шаге. Кроме базы этот самый "теневой регистр" содержит еще и лимит, и права доступа. Еще раз: как только мы загружаем в видимую часть (в селектор) соответствующее значение, процессор сразу же по селектору (а конкретно - по полю индекс) выпасает базу, лимит и права доступа из дескриптора и заносит их в "теневую часть" сегментного регистра, тем самым облегчая себе жизнь в дальнейшем.

Если мы вдруг решим неожиданно поменять значение базы в дескрипторе для какого-либо сегмента, селектор которого в данный момент уже находится в сегментном регистре, то мы также должны позаботиться и о перезагрузке сегментного регистра, т.к. в теневой части останутся старые значения базы и лимита, и фактически процессору абсолютно наплевать на то, что творится в таблице

дескрипторов, он руководствуется только текущим значением базы и лимита в теневой части. Таким образом, если в дескрипторе были изменены база или лимит, сегментный регистр нужно перезагрузить.
Загрузить сегментные регистры (явно или неявно) позволяют следующие команды:
ЯВНО:
MOV
POP - значение из стека
LDS - загрузить DS
LES - загрузить ES
LSS - загрузить SS
LGS - загурзить GS
LFS - загрузить FS
НЕЯВНО:
CALL, JMP, RET и другие. Чаще всего "неявные" команды изменяют значение именно CS-регистра, но в некоторых случаях и других.
В реальном режиме архитектура процессора никуда не девается, при загрузке селектора в сегментный регистр процессор сам создает соответствующий дескриптор в его скрытой части. Он описывает 16-битный сегмент, начинающийся по указанному адресу с границей 64 Кб.

Переключение в защищенный режим

CR – Control Register. Регистр, который отвечает за управление процессором и режимами его работы. В нем много различных полей, и вдаваться в подробности каждого сейчас не будем, но скажу лишь, что переход в защищенный режим осуществляется установкой бита PE (Protected Enabled, самый младший бит регистра) в значение 1.
Доступ регистру возможен через инструкции MOV CR0, AX; LMSW AX/SMSW AX.