**Защищённый режим.**

**Глава 1. Обзор режимов микропроцессоров IA-32. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_in/main.htm;j=false;s=1280*1024;d=24;rand=0.8648600195916392**

**Процессоры IA-32 могут работать в четырёх режимах: режиме реальных адресов, защищённом режиме, режиме виртуального процессора 8086 и режиме системного управления. Теперь более подробно об этих режимах.**

|  |  |
| --- | --- |
| **1.** | **Режим реальных адресов (real address mode). Также вы могли встречать термин "реальный режим", что должно обозначать то же самое, хотя, это немного не правильно и ещё одно название этого режима, встречающееся в документации Intel: R-Mode. Режим реальных адресов - это режим, в котором процессор работает как "экстишка" (8086), но позволяет пользоваться большинством своих технологий (MMX / SSE / SSE2, 32-разрядные регистры общего назначения, регистры управления и отладки и пр.). После аппаратного сброса процессор переходит в этот режим и начинает выполнять программную инициализацию из BIOS-а.          Вообще говоря, режим реальных адресов в современных процессорах предназначен для запуска компьютера и подразумевается, что операционная система будет работать в защищённом режиме (поэтому оптимизация по производительности для процессоров IA-32 производится для защищённого режима).          В режиме реальных адресов не доступны основные достоинства процессора - виртуальная память, мультизадачность, уровни привилегий, работа с кэшами, буферами TLB, буфером ветвлений и некоторыми другими технологиями, обеспечивающими высокую производительность.** |
| **2.** | **Защищённый режим (protected mode или P-Mode). Как утверждает Intel, это "родной" (native) режим 32-разрядного процессора.          В защищённый режим процессор надо переводить специальными операциями над системными регистрами и войти в этот режим процессор может только из режима реальных адресов. При работе в защищённом режиме процессор контролирует практически все действия программ и позволяет разделить операционную систему, драйвера и прикладные программы разными уровнями привилегий. Благодаря этому ОС может ограничить области памяти, выделяемой программам, запретить или разрешить ввод/вывод по любым адресам, управлять прерываниями и многое другое. При попытке программы выйти за допустимый диапазон адресов памяти, выделенной ей, либо при обращении к "запрещённым" для неё портам процессор будет генерировать исключения - специальный тип прерываний. Грамотно оперируя исключениями, операционная система может контролировать действия программ, организовать систему виртуальной памяти, мультизадачность и другие программные технологии.          В защищённом режиме максимально доступны все ресурсы процессора. Например, R-Mode максимальный диапазон адресов памяти ограничен одним мегабайтом, а в защищённом режиме он расширен до 4 Гб для процессоров 386 и 486 и 64 Гб для Pentium-ов.** |
| **3.** | **Режим виртуального процессора 8086 (virtual-8086 mode) или V-Mode, или "виртуальный режим" (что, вообще говоря, не совсем правильно).          В V-Mode можно войти только из защищённого режима. В этом режиме процессор эмулирует работу 8086 процессора (1 Мб адресного пространства, "обычные" прерывания и пр.), но при этом процессор сохраняет все средства контроля, присущие защищённому режиму.          Обычно, V-Mode используется в операционных системах для запуска программ, рассчитанных на процессор 8086 (так называемая, "обратная совместимость ПО"). V-Mode реализуется как отдельная задача в мультизадачной среде и она может взаимодействовать с другими задачами, если, конечно же, ей позволит это операционная система.** |
| **4.** | **Режим системного управления (system management mode или S-Mode).** |

**Глава 2. Уровни привилегий. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_in/chap_1.htm;j=false;s=1280*1024;d=24;rand=0.6162536588782274**

**Защищённый режим (protected mode или P-Mode) обладает некоторыми особенностями, о которых нужно знать прежде, чем вы будете его использовать.   
        При работе в защищённом режиме процессор следит за правильным выполнением текущей программой ряда условий, например, программа не должна обращаться по определённым адресам портов ввода/вывода, запрещённых для неё. Если всё же происходит нарушение какого-либо условия, то процессор генерирует специальный тип прерывания, так называемое исключение, снабжает это прерывание информацией, описывающей где произошло нарушение и как оно произошло. Далее, специальная процедура обрабатывает это прерывание и решает, что дальше делать с "виноватой" программой (например, прекратить её выполнение).   
        Определением условий должна заниматься операционная система. Когда программа переводит процессор в защищённый режим, то ей, как говорится, "можно всё". Сразу после входа в защищённый режим процессор позволяет программе устанавливать свои условия для самой себя и для других программ. Для того, чтобы эти условия не могла переопределить другая программа, в процессоре введена система уровней привилегий. Благодаря этому, операционная система, например, может разрешить работу с дисковыми накопителями только для себя и тогда вирусы будут бессильны - они не смогут обратиться к дискам через порты ввода/вывода (разве что, только через "дыры" в самой операционной системе).   
        Основой защищённого режима являются уровни привилегий. Уровень привилегий - это степень использования ресурсов процессора. Всего таких уровней четыре и они имеют номера от 0 до 3. Уровень номер 0 - самый привелигерованный. Когда программа работает на этом уровне привилегий, ей "можно всё". Уровень 1 - менее привелигерованный и запреты, установленные на уровне 0 действуют для уровня 1. Уровень 2 - ещё менее привелигерованный, а 3-й - имеет самый низкий приоритет. Таким образом, оптимальная схема работы программ по уровням привилегий будет следующая:**

* **уровень 0: ядро операционной системы,**
* **уровень 1: драйвера ОС,**
* **уровень 2: интерфейс ОС,**
* **уровень 3: прикладные программы.**

**Разумеется, это не единственный способ. Можно определить всю работу процессора в нулевом уровне - и ядро ОС, и драйвера и прикладные программы; можно, например, не разделять драйвера от ядра и т.п., но тогда в такой системе не будет реализован встроенный в процессор механизм защиты программ и данных друг от друга и система будет неустойчивой.**

**Уровни привилегий 1, 2 и 3 подчиняются условиям, установленным на уровне 0, поэтому функционально эти четыре уровня можно разделить на 2 группы: уровень привилегий системы (0) и уровни пользователя (1, 2 и 3). На первый взгляд кажется, что проще было бы реализовать всего два уровня привилегий - системный и пользовательский, но со временем вы обнаружите, что четыре уровня привилегий - это очень удобно и гораздо лучше двух.   
        Программы и данные ограничены внутри своих уровней привилегий. Например, если программа работает на уровне привилегий 2, то она не сможет передать управление процедуре, работающей на любом другом уровне (0, 1 и 3), также, она не сможет обратиться к данным, определённым для использования на других уровнях привилегий. Процессор не допустит этого и в случае нарушений привилегий при доступе к данным и коду сгенерирует исключение общей защиты (general-protection exception) и передаст управление операционной системе, чтобы она приняла меры к нарушителю.   
        Сам по себе уровень привилегий ещё ничего не значит, его нельзя "установить в процессоре". Уровень привилегий применяется как одно их свойств при описании различных объектов, например, сегмента кода и действует при работе только с этим объектом.   
        Уровень привилегий обозначается как "PL" (от Privilege Level) и применяется в сочетаниях, например, IOPL - Input/Output Privilege Level - уровень привилегий ввода/вывода.**

**Когда процессор переходит в защищённый режим, то подразумевается, что программа будет работать в нулевом уровне привилегий.**

**Глава 3. Адресация памяти в защищённом режиме. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_in/chap_2.htm;j=false;s=1280*1024;d=24;rand=0.17705337984761105**

**Первое, с чем сталкивается программа при переходе в защищённый режим - это совершенно другая система адресации памяти. Для начала, давайте вспомним, как это происходит в режиме реальных адресов - для обращения к памяти используется пара 16-разрядных регистров - сегментный регистр и смещение.   
        В сегментном регистре находится адрес сегмента. Сегмент - это область памяти размером в 64 Кб, которая должна начинаться на границе параграфа или, другими словами, на 16-байтной границе, то есть сегмент может начинаться только по адресу 0, 16, 32, 48 и т.д. Адресное пространство процессора 8086 равно одному мегабайту - это адреса в диапазоне от 00000h до FFFFFh. Т.к. сегмент начинается по адресу, кратному 16 (10h), то возможные адреса начала сегмента будут 00000h, 00010h, 00020h, ... , FFFF0h. Как видите, информация содержится в старших четырёх шестнадцатеричных разрядах - их и хранят в сегментном регистре. Другими словами, можно взять значение из сегментного регистра, умножить его на 16 (т.е. на 10h) и вы получите адрес начала сегмента.   
        Для указания конкретного адреса внутри сегмента используется второй 16-разрядный регистр, так называемое смещение. Использование пары регистров сегмент:смещение обеспечивает доступ ко всему мегабайту адресного пространства.   
        Каждый раз, перед тем как процессор обратится к памяти по адресу, указанному в паре регистров сегмент:смещение, он вычисляет адрес памяти по следующей схеме:**

**физический\_адрес = сегмент \* 10h + смещение.**

**Осталось заметить, что, вообще говоря, в режиме реальных адресов адресное пространство немного больше одного мегабайта, а именно:**

**1Мб + 64Кб - 16 байт.**

**Такой адрес получается, если загрузить в пару регистров максимальные значения:**

**FFFF:FFFF**

**При этом адрес будет равен:**

**FFFFh \* 10H + FFFFh = FFFF0h + FFFFh = 10FFEFh = 1114095**

**Теперь давайте рассмотрим схему образования адреса в защищённом режиме, как это происходит в процессоре i386 (80386 - это базовая модель для любого Pentium-а).   
        Адрес памяти в 32-разрядном процессоре является также 32-разрядным. Это значит, что адресное пространство для такого процессора равно 4 Гб (2 32 байт).   
      Адресация памяти также производится через сегмент и смещение в сегменте, для чего используется пара регистров, но для описания сегмента используется больше информации, а именно:**

* **32-разрядный адрес,**
* **20-разрядный предел сегмента (предел = размер - 1),**
* **номер уровня привилегий сегмента,**
* **тип сегмента (код, стек, данные или системный объект)**
* **и дополнительные свойства, о которых подробнее будет сказано в других главах.**

**Как видите, информации об одном сегменте достаточно много. Для того, чтобы её хранить, используется не регистр, а специальная область памяти. Сегмент по-прежнему указывается в сегментном регистре, но теперь в нём хранится номер сегмента из списка определённых сегментов. Этот номер называется селектор.   
        В качестве смещения используется 16- или 32-разрядный регистр.   
        При обращении к памяти, процессор проверяет возможность доступа к сегменту по уровню привилегий, проверяет, не превысил ли адрес предел сегмента и можно ли обращаться к этому сегменту в данном случае (например, запрещена передача управления в сегмент, описывающий данные или стек). Если в результате проверки будет обнаружено нарушение какого-либо условия, то процессор сгенерирует исключение и тем самым обеспечит защиту.**

**Предел сегмента - это максимально допустимое смещение внутри него, таким образом, предел сегмента определяет его размер: размер\_сегмента = предел\_сегмента + 1**

**Обратите внимание, что значение предела - 20-разрядная величина, это значит, что максимальное значение предела равно 2 20 - 1.   
        Процессор измеряет размер сегмента двумя типами величин: либо байтами, либо страницами. Страница - это блок памяти размером в 4Кб.   
        В описании сегмента можно указать, в каких единицах измеряется сегмент и тогда можно получить два типа сегментов с максимальными размерами:**

** в 1Мб ( 2 20 байт ) или**

** в 4Гб ( 2 20 страниц = 2 20 \* 4Кб = 2 20 \* 2 12 = 2 32 байт )**

**Эта способность измерять сегмент либо байтами, либо страницами, называется гранулярность.   
        Значение предела сегмента может быть любым, от 0 до 2 20 - 1, гранулярность устанавливается по усмотрению программиста и может быть либо байтная, либо страничная. Всё это позволяет определять сегменты любого размера - от 0 байт до 4Гб.**

**Глава 4. Дескриптор. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_in/chap_3.htm;j=false;s=1280*1024;d=24;rand=0.018340058767883693**

**Прежде чем программа сможет обратиться по какому-либо адресу памяти, она должна определить набор сегментов, через которые она сможет получить доступ к памяти.   
       Сегмент определяется в виде структуры данных, которая называется дескриптор. Размер дескриптора - 8 байт, все дескрипторы хранятся последовательно в специально отведённой области памяти - глобальной дескрипторной таблице.   
       Далее приведен формат дескриптора:**

**биты:**

**0..15: предел, биты 0..15**

**16..31: адрес сегмента, биты 0..15**

**32..39: адрес сегмента, биты 16..23**

**40: бит A (Accessed)**

**41..43: Тип сегмента**

**44: бит S (System)**

**45,46: DPL (Descriptor Privilege Level)**

**47: бит P (Present)**

**48..51: предел, биты 16..19**

**52: бит U (User)**

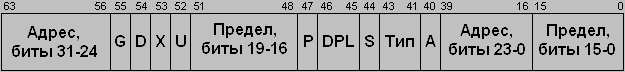
**53: бит X (reserved)**

**54: бит D (Default size)**

**55: бит G (Granularity)**

**56..63: адрес сегмента, биты 24..31**

**Эта структура поясняется на рис. 4-1:**

**  
Рисунок 4-1. Структура дескриптора.**

**Как видите, значения предела и адреса сегмента "разбросаны" по всей структуре дескриптора. Это объясняется тем, что впервые защищённый режим появился в 16-разрядном процессоре 80286 и для совместимости с ним дескриптор не переделывали, а расширили дополнительными полями (биты с 49 по 63), благодаря чему программы, написанные для защищённого режима 286-го процессора работают и на 32-разрядных процессорах.   
        Практически, в программах формат дескриптора удобнее использовать в следующем виде:**

**descriptor:**

**dw limit\_low ; младшее слово предела**

**dw address\_low ; младшее слово адреса**

**db address\_hi ; 3-й (из четырёх) байт адреса**

**db access\_rights ; права доступа**

**db limit\_hi\_and\_flags ; старшая часть предела и флаги GDXU**

**db address\_hi ; 4-й байт адреса**

**Байт прав доступа имеет следующий формат:**

**биты:**

**0: бит A (Accessed)**

**1..3: тип сегмента**

**4: бит S (System)**

**5,6: поле DPL**

**7: бит P (Present)**

**Байт старшей части предела и флагов GDXU имеет формат:**

**биты:**

**0..3: старшая часть предела (биты 16..19)**

**4: бит U**

**5: бит X**

**6: бит D**

**7: бит G**

**Элементы дескриптора.**

|  |  |
| --- | --- |
| **** | **Адрес сегмента - также называется базовым адресом, - 32-разрядный адрес области памяти, с которой начинается сегмент.** |
| **** | **Предел сегмента - предельное значение смещения в сегменте; также можно рассматривать предел как размер сегмента минус один элемент размера - байт или страницу, смотря в чём измеряется сегмент.** |
| **** | **Бит A (Acessed) - бит доступа в сегмент. Этот бит показывает, был ли произведен доступ к сегменту, описываемому этим дескриптором, или нет. Если процессор обращался к сегменту для чтения или записи данных или для выполнения кода, размещённых в нём, то бит A будет установлен (равен 1), иначе - сброшен (0).      С помощью бита A операционная система может определить, использовался ли за последнее время этот сегмент или нет и предпринять какие-либо действия.      Бит A процессором только устанавливается, сбрасывать его должна операционная система.      При создании нового дескриптора подразумевается, что бит A будет равен 0 (т.е. обращений к этому сегменту ещё не было).** |
| **** | **Тип сегмента - трёхбитовое поле, определяющее тип сегмента (см. таблицу 4-1). Каждый бит типа сегмента имеет следующие значения:**  ** старший бит (3-й бит в байте прав доступа):**  **если 0, то это сегмент данных, если 1 - то кода**  ** средний бит (2-й бит) - W (write-enable) бит разрешения записи:**  **если 0, то запись запрещена, 1 - разрешена**  ** младший бит (1-й бит) - E (expansion-direction) направление расширения сегмента:**  **если 0, то вверх (в сторону старших адресов) - как обычно,**  **если 1, то вниз (в сторону младших адресов) - как в стеке.**  **Для сегмента кода значения битов W и E интерпретируются несколько иначе (см. таблицу 4-1)** |

**Таблица 4-1. Тип сегмента**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Бит #** | **11** | **10** | **9** | **Тип** |  |
| **Название** |  | **E** | **W** |
|  | **0** | **0** | **0** | **Данные** | **Только чтение** |
| **0** | **0** | **1** | **Данные** | **Чтение и запись** |
| **0** | **1** | **0** | **Данные** | **Только чтение, расширяется вниз** |
| **0** | **1** | **1** | **Данные** | **Чтение и запись, расширяется вниз** |
|  | | | | | |
|  | **1** | **0** | **0** | **Код** | **Только выполнение** |
| **1** | **0** | **1** | **Код** | **Только выполнение** |
| **1** | **1** | **0** | **Код** | **Только выполнение, согласованный** |
| **1** | **1** | **1** | **Код** | **Выполнение и считывание, согласованный** |

**Примечания:**

|  |  |
| --- | --- |
| **1.** | **В защищённом режиме процессор запрещает запись в сегмент кода и, как видно из таблицы, не всегда разрешает даже простое считывание (хотя, запись в сегмент кода, конечно же, можно сделать, но не явно)** |
| **2.** | **Согласованный сегмент кода будет обсуждаться в следующих главах.** |

** Бит S (System) - определяет системный объект. Если этот бит установлен, то дескриптор определяет сегмент кода или данных, а если сброшен, то системный объект (например, сегмент состояния задачи, локальную дескрипторную таблицу, шлюз).**

** Поле DPL (Descriptor Privilege Level) - Уровень привилегий, который имеет объект, описываемый данным дескриптором. Это двухбитовое поле, в него при создании дескриптора записывают значения от 0 до 3, определяющее уровень привилегий.   
    Например, Если вам нужно, чтобы процессор выполнял программу на нулевом уровне привилегий, то в DPL дескриптора сегмента кода, где размещается программа, должно быть значение 00B.**

** Бит P (Present) - Присутствие сегмента в памяти. Если этот бит установлен, то сегмент есть в памяти, если сброшен, то его нет. Этот бит применяется при реализации механизма виртуальной памяти - если программе понадобится память, то она сохранит содержимое какого-либо сегмента на диск и сбросит бит P. Если любая программа в дальнейшем обратится к этому сегменту, то процессор сгенерирует исключение неприсутствующего сегмента и запустит обработчик этой ситуации, который должен будет подгрузить содержимое сегмента с диска и установить бит P. После этого управление снова передаётся команде, обратившейся к этому сегменту (производится повторное выполнение команды, вызвавшей сбой) и работа программы будет продолжена. Бит P устанавливается и сбрасывается программами, сам процессор его только считывает.**

** Бит U (User) - Бит пользователя. Этот бит процессор не использует и позволяет программе использовать его в своих целях.**

** Бит X (rezerved) - Зарезервированный бит. Intel не рекомендует использовать этот бит, так как он может понадобится в более поздних моделях процессоров.**

** Бит D (Default size) - Размер операндов по умолчанию. Если бит сброшен, по процессор использует объект, описываемый данным дескриптором, как 16-разрядный, если бит установлен - то как 32-разрядный. Если ваша программа имеет 32-разрядный код, то он должен размещаться в 32-разрядном сегменте кода (т.е. в дескрипторе такого сегмента бит D должен быть равен 1). В защищённом режиме допускается использование одновременно 16- и 32-разрядных сегментов, но при написании новых программ подразумевается, что все сегменты будут 32-разрядные. Подробнее об этом см. в главе "Смешивание 16- и 32-разрядного кода".**

** Бит G (Granularity) - Гранулярность сегмента, т.е. единицы измерения его размера. Если бит G=0, то сегмент имеет байтную гранулярность, иначе - страничную (одна страница - это 4Кб).   
    Например, сегмент, имеющий предел, равный 2, при G=0 будет иметь размер в три байта, а при G=1 - 12Кб (3 страницы)**

**Создавать дескрипторы достаточно легко. Все биты и битовые поля находятся в байте прав доступа access\_rights (P, DPL, S, Тип, A ) и в старших четырёх разрядах байта limit\_hi\_and\_flags (G, D, X, U). Дескрипторы по типу отличаются значениями байта прав доступа, по свойствам - битами G и D, остальное - значения базового адреса и предела. Когда мы будем рассматривать пример перехода в защищённый режим, вы увидите один из вариантов конструирования дескрипторов.**

**Глава 5. Селектор. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_in/chap_4.htm;j=false;s=1280*1024;d=24;rand=0.9319043013295819**

**При адресации памяти в защищённом режиме команды ссылаются на сегменты, указывая не их адреса (как в режиме реальных адресов), а описания сегментов (их дескрипторы). Указатель на описание сегмента называется селектор. Другими словами, селектор - это номер дескриптора из таблицы дескрипторов.   
        Адресация производится через пару регистров сегмент:смещение, причём, в качестве сегментного регистра используются обычные CS, SS, DS, ES, FS и GS (последние два появились в 386-м процессоре), но в них указывается не адрес сегмента, а селектор дескриптора.   
        Селекторы нужны, по крайней мере, по трём причинам:**

|  |  |
| --- | --- |
| **1.** | **Описание сегмента занимает 8 байт и использовать 8-байтные сегментные регистры было бы крайне неэффективно.** |
| **2.** | **Селекторы имеют размер в 16 бит, благодаря чему их можно использовать в сегментных регистрах и обращаться к памяти можно по-прежнему через пару регистров.** |
| **3.** | **Параметры всех сегментов хранятся в отдельной области памяти, доступ к которой имеет только операционная система. Программа, используя селектор, сможет получить о сегменте совсем немного информации и не сможет изменить параметры сегмента, благодаря чему очень удачно реализуется механизм защиты.** |

**Адрес памяти можно указывать не только через пару регистров, но и в переменных, через пару значений селектор:смещение.**

**Можно было бы определить селектор просто, как номер сегмента, но в защищённом режиме кроме сегментов, дескриптор может определять целый ряд других системных объектов (например, задач), поэтому лучше не упрощайте понятия селектора и дескриптора, а постарайтесь привыкнуть к этой терминологии.**

**Селектор имеет следующий формат:**

**http://sasm.narod.ru/docs/pm/pm_img/pm_in5_1.gif  
Рисунок 5-1. Формат селектора.**

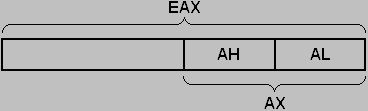
* **Двухбитовое поле RPL (Requested Privilege Level) содержит номер уровня привилегий, которое имеет текущая программа. Значение этого поля процессор использует для защиты по привилегиям. К одному и тому же дескриптору можно обращаться, используя селекторы с разными значениями RPL, но процессор позволит доступ только при определённых условиях (подробнее об этом см. в главе "Защита по привилегиям").**
* **Бит TI (Table Indicator) определяет таблицу, из которой выбирается нужный дескриптор. Если бит TI = 0, то обращение производится к глобальной дескрипторной таблице GDT (она одна на всю систему), если TI = 1 - то к текущей локальной дескрипторной таблице LDT (таких может быть много).   
      Подробнее дескрипторные таблицы обсуждаются в соответствующих главах.**
* **Index - это собственной номер дескриптора, от 0 до 8191. Т.к. поле индекса состоит из 13 бит, то максимальное число дескрипторов, одновременно существующих в системе равно 2 13, т.е. 8192. Как видите, это довольно-таки много и вполне удовлетворяет любым системным запросам. На самом деле, число дескрипторов можно значительно увеличить за счёт использования множества дополнительных локальных дескрипторных таблиц.**

**Использование селекторов достаточно простое. Для тех дескрипторов, которые будут определены заранее, например, сегментов кода, стека и данных, селекторы подготавливаются как константы и затем используются для загрузки в сегментные регистры. Для дескрипторов, которые программа будет динамически создавать, селекторы придётся определять в переменных и загружать в сегментные регистры из памяти либо конструировать "на ходу", или даже как константы - всё зависит от конкретных условий. Способы использования селекторов и дескрипторов вы можете найти в примерах, которые будут следовать в дальнейших главах.**

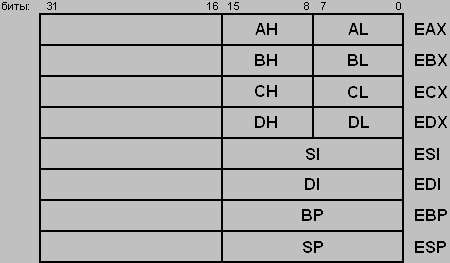
**Обращение к дескрипторной таблице процессор производит только в момент загрузки в сегментный регистр нового селектора. После этого содержимое дескриптора копируется в так называемый "теневой регистр", к которому имеет доступ только сам процессор и из которого оно в дальнейшем используется. Любое последующее обращение к сегменту будет происходить с помощью теневого регистра, без обращения к дескрипторной таблице и не потребует лишних тактов на циклы чтения памяти. Правда, эти такты тратятся каждый раз, когда вы загружаете новый селектор, но это не высокая плата за защиту дескрипторов от недозволенного доступа.   
        При загрузке недопустимого значения селектора процессор будет генерировать исключение, даже если вы не обращались через него к памяти.**

**Глава 6. Использование регистров общего назначения для адресации. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_in/chap_5.htm;j=false;s=1280*1024;d=24;rand=0.9160302879666462**

**Для обращения к памяти через регистры обычно используются четыре 16-разрядных регистра общего назначения (РОН): BX, SI, DI и BP. В защищённом режиме для адресации можно использовать все 8 регистров общего назначения.   
        В 32-разрядных процессорах обычные регистры расширены до 32-х разрядов. Вот так регистр AX расширен до регистра EAX:**

**  
Рисунок 6-1. Формат регистра EAX.**

**Как видите, регистр AX является составной частью регистра EAX (так же, как регистры AL и AH являются составными частями регистра AX), т.е. если вы обращаетесь к регистру AX, то вы меняете содержимое регистра EAX.   
        Подобным образом расширены все 8 регистров общего назначения:**

**  
Рисунок 6-2. Регистры общего назначения.**

**Также, расширен 16-разрядный регистр FLAGS - теперь это 32-разрядный EFLAGS, младшая половина которого представляет собой FLAGS:**

**http://sasm.narod.ru/docs/pm/pm_img/pm_in6_3.gif  
Рисунок 6-3. Формат регистра EFLAGS.**

**Регистр IP (Instruction Pointer) был расширен до 32-разрядного EIP:**

**http://sasm.narod.ru/docs/pm/pm_img/pm_in6_4.gif  
Рисунок 6-4. Формат регистра EIP.**

**Регистр EIP непосредственно использовать нельзя, но теперь следует учитывать, что при 32-разрядной адресации памяти в качестве адреса перехода можно указывать 32-разрядную величину.   
        Хотя указатель стека (регистр ESP) также относится к регистрам общего назначения и может использоваться в командах, настоятельно рекомендуется никогда не привлекать его к использованию вне стека. Особенно это важно при работе в защищённом режиме, когда процессор автоматически использует текущее значение стека, чтобы поместить в него значения, например, при обработке исключений.**

**В 32-разрядном процессоре вы по-прежнему можете адресовать память через четыре 16-разрядных регистра BX, SI, DI и BP, но дополнительно к этому можно использовать каждый из 32-разрядных регистров общего назначения, причём в любом режиме (не только защищённом). Например:**

**mov ax,[ ebx ] ; Поместить в AX значение из памяти**

**; по адресу DS:EBX**

**mov dx,[ ecx ] ; Поместить в DX значение из памяти**

**; по адресу DS:ECX**

**mov cx,es:[ eax ] ; Поместить в CX значение из памяти**

**; по адресу ES:EAX**

**Дополнительно к этой возможности введены следующие:**

* **Использование константы и регистра:**
* **mov eax,[ ecx + 1 ]**
* **mov bl,[ edx + 12345678h ]**
* **Сумма двух регистров:**
* **mov ebp,[ ebx + edi ]**
* **mov eax,[ ecx + edx ]**
* **Сумма двух регистров и константы:**
* **mov bl,[ edx + eax + 12345678h ]**
* **Масштаб - автоматическое умножение на 2, 4 или 8 одного из регистров, участвующих в образовании адреса:**
* **mov ax,[ ebx \* 2 ]**
* **mov cl,[ edx + ebp \* 4 ]**
* **mov esi,[ edi + eax \* 8 + 12345678h ]**

**Очевидно, что возможности, которые нам предоставляла XT, 32-разрядные процессоры значительно расширили. Следует, однако, учитывать, что в ранних процессорах (i386 и i486) на вычисление эффективного адреса процессор расходует дополнительное время, но благодаря универсальности этой системы адресации всё равно имеет место итоговый выигрыш в производительности.**

**При использовании 32-разрядных регистров для адресации в режиме реальных адресов, следует учитывать, что размер сегмента фиксирован и равен 64 Кб. Если процессор сформирует адрес, больший 64 Кб, то процессор зависнет, т.е. он не будет производить заворачивание адресов. Например:**

**mov eax,1234h**

**mov bl,[ eax ] ; В регистр BL будет произведена загрузка**

**; значения с адреса DS:EAX, равного DS:1234h.**

**mov edx,ffffh**

**mov bl,[ eax + edx ] ; Эффективный адрес будет равен**

**; 1234h + ffffh = 11233h (это больше,**

**; чем 64 Кб. Процессор зависнет.**

**Использование 32-разрядных регистров для адресации памяти в защищённом режиме очень распространено, в основном, из-за того, что размер сегментов может достигать 4 Гб, да и просто потому, что это удобно.**

**Глава 7. Глобальная дескрипторная таблица. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_in/chap_6.htm;j=false;s=1280*1024;d=24;rand=0.38544542560375195**

**Прежде, чем процессор перейдёт в защищённый режим, должна быть определена глобальная дескрипторная таблица GDT (Global Descriptor Table), так как все сегменты и прочие системные объекты должны быть описаны в дескрипторной таблице.   
        Глобальная дескрипторная таблица GDT - это область памяти, в которой находятся дескрипторы. Процессору всё равно, где именно вы расположили эту таблицу, но в любом случает она будет находится в первом мегабайте адресного пространства, потому что только из режима реальных адресов можно перевести процессор в защищённый режим. Также подразумевается, что сама таблица GDT будет выравнена на границу 8 байт, так как дескрипторы, из которых она состоит, имеют 8-байтный размер. Такое выравнивание позволит процессору максимально быстро обращаться к дескрипторам, что, естественно, увеличивает производительность.   
        Число дескрипторов, определённых в GDT, может быть любым, от 0 до 8192. Нулевой дескриптор, т.е. определённый в самом начале GDT, процессор не использует, обращение к такому дескриптору могло бы быть, когда поле Index селектора равно 0. Если всё же в программе встречается обращение к нулевому дескриптору, то процессор генерирует исключение и не позволит доступ к такому дескриптору. В связи с этим, везде в литературе рекомендуется использовать нулевой дескриптор как шаблон, на основе которого программа может создавать новые дескрипторы, но на практике их удобнее создавать иными способами, о которых мы ещё будем говорить.**

**GDT используется процессором всё время, пока он находится в защищённом режиме. Параметры GDT хранятся в специальном 48-разрядном регистре GDTR:**

**http://sasm.narod.ru/docs/pm/pm_img/pm_in7_1.gif  
Рисунок 7-1. Формат регистра GDTR.**

**Формат регистра GDTR следующий:**

|  |  |
| --- | --- |
| **биты:** |  |
| **0..15:** | **16-разрядный предел GDT** |
| **15..47:** | **32-разрядный адрес начала GDT** |

**Адрес начала GDT - это тот адрес, по которому вы разместили GDT.   
        Предел таблицы GDT - это максимальное смещение относительно её начала.**

**Например, вы создаёте GDT, состоящую из 3-х дескрипторов - для сегментов кода, стека и данных. Общее число дескрипторов будет равно четырём, потому что первым по счёту будет идти нулевой дескриптор, а за ним уже остальные три:**

|  |  |
| --- | --- |
| **Смещение от начала GDT** | **Назначение дескриптора** |
| **0** | **Нулевой** |
| **8** | **Сегмент кода** |
| **16** | **Сегмент стека** |
| **24** | **Сегмент данных** |

**Размер GDT в данном случае будет равен 32 байтам, следовательно, предельное смещение в таблице будет равно 31 - это и есть предел GDT.**

**Для загрузки значения в регистр GDTR используется команда LGDT. Операндом этой команды является 48-разрядное значение адреса в памяти, где размещается адрес и предел GDT. Вы также можете сохранить содержимое GDTR командой SGDT, указав в операнде адрес 48-разрядной переменной в памяти.**

**Далее приводится пример подготовки параметров GDT и их загрузки в регистр GDTR.**

**Пример 1. Подготовка параметров GDT и их загрузка в регистр GDTR.**

**Алгоритм:**

**1. Вычисляем 32-разрядный адрес GDT**

**2. Вычисляем размер GDT**

**3. Сохраняем параметры GDT в 48-разрядную переменную**

**4. Загружаем значение в регистр GDTR**

**Код:**

**; 1. Вычисляем 32-разрядный адрес GDT**

**xor eax,eax ; EAX = 0; адрес будем вычислять в**

**; регистре EAX.**

**mov ax,ds ; Подразумевается, что GDT находится в**

**; текущем сегменте данных.**

**shl eax,4 ; EAX = адрес начала сегмента DS**

**xor edx,edx ; EDX = 0.**

**lea dx,GDT ; EDX = DX = смещение начала GDT**

**; относительно DS.**

**add eax,edx ; EAX = полный физический адрес GDT**

**; в памяти.**

**; 2. Вычисляем размер GDT**

**mov cx,4 ; CX = число дескрипторов +**

**; + нулевой дескриптор.**

**shl cx,3 ; CX = CX \* 8 - столько байт будут**

**; занимать в GDT эти дескрипторы.**

**dec cx ; Предел меньше размера на 1**

**; 3. Сохраняем параметры GDT в 48-разрядную переменную**

**mov GDT\_address,eax**

**mov GDT\_limit,cx**

**; 4. Загружаем значение в регистр GDTR**

**lgdt GDT\_params**

**;--------------------------------------------------------------------------**

**GDT\_params label fword**

**GDT\_limit dw ?**

**GDT\_address dd ?**

**GDT:**

**dd ? ; 0-й дескриптор**

**dd ?**

**dd ? ; 1-й дескриптор (код)**

**dd ?**

**dd ? ; 2-й дескриптор (стек)**

**dd ?**

**dd ? ; 3-й дескриптор (данные)**

**dd ?**

**Остаётся добавить, что размер GDT желательно не менять в процессе выполнения программ в защищённом режиме. Если ваша программа будет динамически создавать новые дескрипторы, то размер GDT лучше всего заранее задать достаточно большим, например, 64 Кб (максимальный размер). Однако, следует учитывать, что при обращении процессора к несуществующим дескрипторам, его поведение непредсказуемо, хотя оно, скорее всего, закончится зависанием.**

**Глава 8. Регистры управления. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_in/chap_7.htm;j=false;s=1280*1024;d=24;rand=0.4475778502003782**

**Все 32-разрядные процессоры, начиная с i386, имеют набор системных регистров, предназначенных для использования в защищённом режиме. Среди них есть регистры управления (Control Registers) CR0, CR1, CR2, CR3 и CR4.   
        Регистры управления, в основном, состоят из флагов. Назначение и использование каждого флага достаточно сложно и требует отдельного рассмотрения. Для начала мы рассмотрим только один бит PE регистра CR0, отвечающего за переход процессора в защищённый режим и обратно. Полный список с описаниями регистров CRi приводится в приложении "Регистры управления CRi".   
        Регистры управления предназначены для считывания и записи информации. Они имеют размер в 32 бита и оперировать ими можно только целиком - считали значение целого регистра, изменили нужные биты и записали обратно. Единственная команда, которой позволен доступ к этим регистрам - это MOV, в качестве операнда которой используется 32-разрядный регистр общего назначения.   
        В дальнейших главах мы будем рассматривать переход в защищённый режим и возврат из него в режим реальных адресов, для чего будем использовать только один бит из регистра CR0 - это нулевой бит, который называется PE (Protection Enable). Если установить этот бит в 1, процессор перейдёт в защищённый режим, если сбросить - то в режим реальных адресов.   
        Процессор после аппаратного сброса переходит в режим реальных адресов и бит PE сброшен.   
        Вот примеры использования бита PE:**

**1. Переводим процессор в защищённый режим.**

**mov eax,cr0 ; Копируем в EAX содержимое регистра CR0.**

**or al,1 ; Устанавливаем в копии 0-й бит, который**

**; соответствует 0-му биту CR0, т.е. биту PE.**

**mov cr0,eax ; Записываем в CR0 обновлённое значение.**

**; Процессор перешёл в защищённый режим.**

**... ; 1-я команда программы, которая выполнится**

**; в защищённом режиме.**

**2. Переводим процессор в режим реальных адресов.**

**mov eax,cr0**

**and al,0feh ; Сбрасываем бит PE.**

**mov cr0,eax ; Процессор перешёл в режим реальных адресов.**

**На самом деле, если вы просто выполните переход в защищённый режим, как показано на приведенном выше примере, то процессор зависнет. Для работы в защищённом режиме процессор использует дескрипторы сегментов, вместо их адресов (которые находятся в сегментных регистрах), поэтому прежде, чем перевести процессор в защищённый режим, нужно провести некоторые подготовительные работы.**

**Глава 9. Вход в защищённый режим. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_in/chap_8.htm;j=false;s=1280*1024;d=24;rand=0.6772249370866638**

**Когда программа переводит процессор в защищённый режим, при всём том богатом потенциале, который предоставляют 32-разрядные процессоры, она оказывается практически беспомощной. В основном это связано с тем, что в защищённом режиме совершенно другая система прерываний и воспользоваться ресурсами, предоставляемыми операционной системой, в которой вы запускаете такую программу, невозможно. Более того, недоступными окажутся прерывания BIOS и IRQ. Подробно работа прерываний описана в разделе "Прерывания в защищённом режиме", там же вы найдёте примеры использования прерываний всех типов (программные, аппаратные и исключения), но пока наша программа не сможет ими воспользоваться.   
        В предыдущих главах не раз упоминалась фраза "операционная система", когда шла речь о программе, работающей в защищённом режиме. Дело в том, что после перевода процессора в P-Mode, программа должна определить действия и условия для всех ситуаций, вплоть до того, что определить драйвера (т.е. управляющие процедуры) для таких устройств, как клавиатура, мышь, видеоадаптер, дисковые накопители и даже таймер. Существует два способа реализации таких драйверов - либо написать их самому (что, вообще говоря, не очень сложно), либо обращаться к BIOS-у в режиме виртуального процессора 8086. Оба этих способа будут описаны в соответствующих разделах, где будет подразумеваться, что вы разобрались с разделом "Защищённый режим".**

**Прежде, чем мы перейдём к примеру, давайте определимся с тем, что нам надо сделать:**

|  |  |
| --- | --- |
| **1.** | **Подготовиться к переходу в P-Mode.** |
| **2.** | **Перейти в P-Mode.** |
| **3.** | **Сообщить в программе о переходе в P-Mode.** |

**В третьем пункте мы заставим программу вывести на экран строку "I am in protected mode!!!", после чего программа зациклит процессор (компьютер повиснет). Дело в том, что возврат в режим реальных адресов немного сложнее, чем переход в защищённый режим и это требует дополнительного объяснения, поэтому в этом примере приводится только переход в P-Mode.   
        Вывод на экран будет происходить в текстовом режиме посредством прямой записи в видеопамять, для чего будет описана отдельная процедура - вы увидите, что код, предназначенный для выполнения в защищённом режиме не требует специальных определений (это будет 16-разрядный код; 32-разрядный определяется немного сложнее).   
        Остаётся добавить, что пример должен выполняться из простой операционной системы, например, MS-DOS, работающей в режиме реальных адресов. Если вы попытаетесь запустить программу из-под ОС, работающей в защищённом режиме (например, Windows), то программа не заработает, т.к. процессор уже будет работать в P-Mode и не допустит повторного входа в этот режим.   
        Что касается зависания процессора в конце выполнения нашего примера - так это нормальное явление при отладке программ в защищённом режиме. Этот режим тем и характерен, что допускает работу только одного "хозяина" одновременно, а наша программа, войдя в защищённый режим, как раз и станет таким "хозяином процессора".**

**Хочу обратить ваше внимание на то, что первой командой после перехода в защищённый режим должна быть команда дальнего перехода (far jump), в которой будет указан селектор дескриптора сегмента кода и смещение в этом сегменте. При работе в защищённом режиме процессор может использовать в сегментных регистрах только селекторы существующих дескрипторов, любые другие значения (например, сегментный адрес) использовать нельзя - процессор сгенерирует исключение общей защиты. Тем не менее, при переходе в защищённый режим регистр CS будет содержать сегментный адрес, который использовался в режиме реальных адресов, поэтому выполнение следующей команды, какой бы она ни была, должно было бы привести к генерации процессором исключения. На самом деле этого не происходит, так как эта команда не выбирается из памяти - она уже находится в конвейере процессора (даже в таком процессоре, как i386, есть конвейер) и поэтому вы можете выполнить эту команду.   
        Команда дальнего перехода обязательно очистит конвейер процессора и заставит его обратится к таблице GDT, выбрать оттуда дескриптор, селектор которого указан в адресе команды и начать выборку команд со смещения, также указанного в этом адресе. Это критический момент в работе программы. Если в GDT, селекторе, смещении или самой команде будет обнаружена ошибка, то процессор сгенерирует исключение, а так как систему прерываний мы для него пока не определяли, то он попросту зависнет либо произойдёт сброс - это уже зависит от "железа".   
        Если вы не выполните первой команду дальнего перехода, а другую, которая не изменит содержимое регистра CS (а это - все остальные команды), то процессор произведёт выборку в конвейер новой команды, используя текущие значения CS:IP, а так как в CS содержится не селектор (процессор уже в защищённом режиме!), то произойдёт исключение и зависание.   
        Это - теория. Такие условия перехода в P-Mode рекомендует Intel и опыт показывает, что лучше придерживаться этих рекомендаций. На практике наблюдаются некоторые чудеса. Например, можно сделать переход не дальний, а короткий (без смены значения в CS) - и программа будет работать, мало того, можно даже не определять стек - всё равно программа работает - процедуры вызываются, можно оперировать стеком, вот только нет уверенности, что стек будет отображаться на ту же область памяти, что была до перехода.   
        С регистрами данных ситуация обстоит хуже - DS можно не инициализировать, но при работе через него вы получите совсем не те данные, что должны были бы, а обращение к ES, отображённому на видеопамять подвешивает процессор.   
        Эта ситуация была обнаружена при тестировании примера 2 на процессоре 80386 DX-40 и причина, на мой взгляд, в ошибках архитектуры, допущенных при проектировании этого процессора. Сам Intel подобные ошибки называет errata и сообщает обо всех обнаруженных багах в процессорах, самую свежую информацию о них вы можете найти на www.intel.com и www.intel.ru.**

**Использование ошибок процессоров не представляется мне практичным, т.к. вариаций одной и той же модели процессоров много - десятки, хотя если вы сможете найти достойное применение ошибкам, то я буду рад опубликовать на сайте ваши статьи.**

**И всё же, возвращаясь к примеру, давайте определимся: мы изучаем "правильное" программирование 32-разрядных процессоров, при этом не используя никаких ошибок в архитектуре и никаких недокументированных особенностей и команд. Это нам обеспечит уверенность в том, что наши программы будут надёжно работать на любых интеловских процессорах и их клонах.**

**Прежде, чем мы перейдём к примеру программы, давайте определим две функции, которые мы будем в дальнейшем использовать.**

**В программе нам понадобится создавать 4 дескриптора - для кода, данных (где будет хранится выводимая строка), стека (он будет использоваться для вызова функции вывода текста) и видеопамяти. Все дескрипторы сегментов подобны друг другу, поэтому удобно их будет создавать отдельной функцией. Исходный код для этой функции (и для второй) поместите как макросы в отдельный файл - мы их будем использовать в дальнейшем и просто подключать к исходникам.**

**Функция первая, "set\_descriptor" предназначенная для создания дескриптора, приводится сразу в том виде, в каком она будет находится в подключаемом файле "pmode.lib".**

**При вызове этой функции подразумевается, что в паре регистров DS:BX находится указатель на текущую позицию в GDT. Функция после создания дескриптора переведёт указатель на позицию для следующего дескриптора. Все необходимые параметры передаются через регистры и функция всего лишь "перетасовывает" их в нужном порядке.**

**Файл "pmode.lib":**

**;-------------------------------------------------------------------------**

**init\_set\_descriptor macro**

**set\_descriptor proc near**

**; Создаёт дескриптор.**

**; DS:BX = дескриптор в GDT**

**; EAX = адрес сегмента**

**; EDX = предел сегмента**

**; CL = байт прав доступа (access\_rights)**

**push eax**

**push ecx ; Регистры EAX и ECX мы будем использовать.**

**push cx ; Временно сохраняем значение access\_rights.**

**mov cx,ax ; Копируем младшую часть адреса в CX,**

**shl ecx,16 ; и сдвигаем её в старшую часть ECX.**

**mov cx,dx ; Копируем младшую часть предела в CX.**

**; Теперь ECX содержит младшую часть**

**; дескриптора (т.е. первые 4 байта -**

**; см. рис. 4-1).**

**mov [ bx ],ecx ; Записываем младшую половину дескриптора в GDT.**

**shr eax,16 ; EAX хранит адрес сегмента, младшую часть**

**; которого мы уже использовали, теперь будем**

**; работать со старшей, для чего сдвигаем её в**

**; младшую часть EAX, т.е. в AX.**

**mov cl,ah ; Биты адреса с 24 по 31**

**shl ecx,24 ; сдвигаем в старший байт ECX,**

**mov cl,al ; а биты адреса с 16 по 23 - в младший байт.**

**pop ax ; Возвращаем из стека в AX значение**

**; access\_rights**

**mov ch,al ; и помещаем его во второй (из четырёх)**

**; байт ECX.**

**; Всё, дескриптор готов. Старшую часть**

**; предела и биты GDXU мы не устанавливаем и**

**; они будут иметь нулевые значения.**

**mov [ bx + 4 ],ecx ; Дописываем в GDT вторую половину**

**; дескриптора.**

**add bx,8 ; Переводим указатель в GDT на следующий**

**; дескриптор.**

**pop ecx**

**pop eax**

**ret**

**endp**

**endm**

**;-------------------------------------------------------------------------**

**Вторую функцию "putzs" добавьте в файл "pmode.lib" после описания первой. Эта функция предназначена для вывода на экран строки, оканчивающейся нулевым байтом (т.е. байтом, равным 00h). Такая строка везде на этом сайте называется ZS-строка или просто ZS (от Zero-String). Также вам будут встречаться строки другого типа - LS (от Lenght-String), длина которых будет задана первым байтом.   
        Функция "putzs" получила своё название от комбинации слов "Put" и "ZS", по аналогии с похожими Си-функциями. В этой функции сохраняются используемые регистры - в приводимом далее примере это никакого эффекта не вызовет - процессор сразу после вывода зависнет, но для других примеров это будет полезно.**

**putzs proc near**

**; DS:BX = ZS ; ZS = Zero-String - строка, оканчивающаяся**

**; нулевым (00h) байтом.**

**; ES:DI = позиция вывода ; ES описывает сегмент видеопамяти,**

**; DI - смещение в нём.**

**push ax**

**push bx**

**push es**

**push di**

**mov ah,1bh ; В AH будет атрибут вывода - светло-циановые**

**; символы на синем фоне.**

**putzs\_1:**

**mov al,[ bx ] ; Читаем байт из ZS-строки.**

**inc bx ; Переводим указатель на следующий байт.**

**cmp al,0 ; Если байт равен 0,**

**je putzs\_end ; то переходим в конец процедуры.**

**mov es:[ di ],ax ; Иначе - записываем символ вместе с**

**; атрибутом в видеопамять по заданному**

**; смещению - цветной символ появится на**

**; экране.**

**add di,2 ; Переводим указатель в видеопамяти на**

**; позицию следующего символа.**

**jmp putzs\_1 ; Повторяем процедуру для следующего байта**

**; из ZS-строки.**

**putzs\_end:**

**pop di**

**pop es**

**pop bx**

**pop ax**

**ret**

**endp**

**;--------------------------------------------------------------------------**

**Теперь сам пример. Прежде чем приступить к его изучению, хочу сделать следующие замечания:**

|  |  |
| --- | --- |
| **1.** | **Программа протестирована и работает, поэтому, если она у вас не заработает, проверьте, ВЫ не ошиблись? Если нет - пишите, разберёмся.** |
| **2.** | **Программа протестирована как отдельный .com-файл. В принципе, вы её можете без изменений перенести в образ .exe-файла или встроить в программу языка высокого уровня - всё должно работать.** |
| **3.** | **Обратите внимание, что таблица GDT и сегменты не выровнены в памяти и программа всё равно работает. Это сделано специально, для демонстрации возможностей P-Mode. Для повышения производительности программы, конечно же следует выровнять все структуры данных, используемые процессором непосредственно (у нас пока это только GDT) и сегменты.** |

**Далее полностью приводится файл "example2.asm", который содержит пример перехода в защищённый режим, включая необходимые для tasm-а атрибуты.**

**Пример 2. Переход в защищённый режим.**

**Файл "example2.asm":**

**include pmode.lib ; Подразумевается, что файл "pmode.lib" находится**

**; в том же каталоге, что и "example2.asm".**

**;-------------------------------------------------------------------------**

**.386p**

**pmode segment use16**

**assume cs:pmode, ds:pmode, es:pmode**

**org 100h**

**main proc far**

**start:**

**;-------------------------------------------------------------------------**

**;-------------------------------------------------------------------------**

**; Определяем селекторы как константы. У всех у них биты TI = 0 (выборка**

**; дескрипторов производится из GDT), RPL = 00B - уровень привилегий -**

**; нулевой.**

**Code\_selector = 8**

**Stack\_selector = 16**

**Data\_selector = 24**

**Screen\_selector = 32**

**;--------------------------------------------------------------------------**

**mov bx,offset GDT + 8 ; Нулевой дескриптор устанавливать**

**; не будем - всё равно он не**

**; используется.**

**xor eax,eax ; EAX = 0**

**mov edx,eax ; EDX = 0**

**push cs**

**pop ax ; AX = CS = сегментный адрес текущего**

**; сегмента кода.**

**shl eax,4 ; EAX = физический адрес начала сегмента кода.**

**; Эта программа, работая в среде операционной системы**

**; режима реальных адресов (подразумевается, что это -**

**; MS-DOS) уже имеет в IP смещение относительно**

**; текущего сегмента кода. Мы определим дескриптор**

**; кода для защищённого режима с таким же адресом**

**; сегмента кода, чтобы при переходе через команду**

**; дальнего перехода фактически переход произошёл**

**; на следующую команду.**

**mov dx,1024 ; Предел сегмента кода может быть любым,**

**; лишь бы он покрывал весь реально**

**; существующий код.**

**mov cl,10011000b ; Права доступа сегмента кода (P = 1,**

**; DPL = 00b, S = 1, тип = 100b, A = 0)**

**call set\_descriptor ; Конструируем дескриптор кода.**

**lea dx,Stack\_seg\_start ; EDX = DX = начало стека (см. саму**

**; метку).**

**add eax,edx ; EAX уже содержит адрес начала сегмента**

**; кода, сегмент стека начнётся с последней**

**; метки программы Stack\_seg\_start.**

**mov dx,1024 ; Предел стека. Также любой (в данном**

**; примере), лишь бы его было достаточно.**

**mov cl,10010110b ; Права доступа дескриптора сегмента**

**; стека (P = 1, DPL = 00b, S = 1,**

**; тип = 011b, A = 0).**

**call set\_descriptor ; Конструируем дескриптор стека.**

**xor eax,eax ; EAX = 0**

**mov ax,ds**

**shl eax,4 ; EAX = физический адрес начала сегмента данных.**

**xor ecx,ecx ; ECX = 0**

**lea cx,PMode\_data\_start ; ECX = CX**

**add eax,ecx ; ECX = физический адрес начала сегмента**

**; данных.**

**lea dx,PMode\_data\_end**

**sub dx,cx ; DX = PMode\_data\_end - PMode\_data\_start (это**

**; размер сегмента данных, в данном примере**

**; он равен 26 байтам). Этот размер мы и**

**; будем использовать как предел.**

**mov cl,10010010b ; Права доступа сегмента данных (P = 1,**

**; DPL = 00b, S = 1, тип = 001, A = 0).**

**call set\_descriptor ; Конструируем дескриптор данных.**

**mov eax,0b8000h ; Физический адрес начала сегмента**

**; видеопамяти для цветного текстового**

**; режима 80 символов, 25 строк**

**; (используется по умолчанию в MS-DOS).**

**mov edx,4000 ; Размер сегмента видеопамяти (80\*25\*2 = 4000).**

**mov cl,10010010b ; Права доступа - как сегмент данных**

**call set\_descriptor ; Конструируем дескриптор сегмента**

**; видеопамяти.**

**; Устанавливаем GDTR:**

**xor eax,eax ; EAX = 0**

**mov edx,eax ; EDX = 0**

**mov ax,ds**

**shl eax,4 ; EAX = физический адрес начала сегмента данных.**

**lea dx,GDT**

**add eax,edx ; EAX = физический адрес GDT**

**mov GDT\_adr,eax ; Записываем его в поле адреса образа GDTR.**

**mov dx,39 ; Предел GDT = 8 \* (1 + 4) - 1**

**mov GDT\_lim,dx ; Записываем его в поле предела образа GDTR.**

**cli ; Запрещаем прерывания. Для того, чтобы прерывания**

**; работали в защищённом режиме их нужно специально**

**; определять, что в данном примере не делается.**

**lgdt GDTR ; Загружаем образ GDTR в сам регистр GDTR.**

**; Переходим в защищённый режим:**

**mov eax,cr0**

**or al,1**

**mov cr0,eax**

**; Процессор в защищённом режиме!**

**db 0eah ; Этими пятью байтами кодируется команда**

**dw P\_Mode\_entry ; jmp far Code\_selector:P\_Mode\_entry**

**dw Code\_selector**

**;--------------------------------------------------------------------------**

**P\_Mode\_entry:**

**; В CS находится уже не сегментный адрес сегмента кода, а селектор его**

**; дескриптора.**

**; Загружаем сегментные регистры. Это обеспечит правильную работу программы**

**; на любом 32-разрядном процессоре.**

**mov ax,Screen\_selector**

**mov es,ax**

**mov ax,Data\_selector**

**mov ds,ax**

**mov ax,Stack\_selector**

**mov ss,ax**

**mov sp,0**

**; Выводим ZS-строку:**

**mov bx,0 ; DS:BX = указатель на начало ZS-строки. Адрес**

**; сегмента данных определён по метке**

**; PMode\_data\_start, а строка начинается сразу после**

**; этой метки, её смещение от метки равно 0,**

**; следовательно, это и будет смещение от начала**

**; сегмента данных.**

**mov di,480 ; Выводим ZS-строку со смещения 480 в**

**; видеопамяти (оно соответствует началу**

**; 3-й строки на экране в текстовом режиме).**

**call putzs**

**; Зацикливаем программу:**

**loop\_1:**

**jmp loop\_1**

**;--------------------------------------------------------------------------**

**init\_set\_descriptor**

**init\_putzs**

**;--------------------------------------------------------------------------**

**; Образ регистра GDTR:**

**GDTR label fword**

**GDT\_lim dw ?**

**GDT\_adr dd ?**

**;--------------------------------------------------------------------------**

**GDT:**

**dd ?,? ; 0-й дескриптор**

**dd ?,? ; 1-й дескриптор (кода)**

**dd ?,? ; 2-й дескриптор (стека)**

**dd ?,? ; 3-й дескриптор (данных)**

**dd ?,? ; 4-й дескриптор (видеопамяти)**

**;--------------------------------------------------------------------------**

**PMode\_data\_start: ; Начало сегмента данных для защищённого режима.**

**;--------------------------------------------------------------------------**

**db "I am in protected mode!!!",0 ; ZS-строка для вывода в P-Mode.**

**;--------------------------------------------------------------------------**

**PMode\_data\_end: ; Конец сегмента данных.**

**;--------------------------------------------------------------------------**

**db 1024 dup (?) ; Зарезервировано для стека.**

**Stack\_seg\_start: ; Последняя метка программы - отсюда будет расти стек.**

**;--------------------------------------------------------------------------**

**main endp**

**pmode ends**

**end start**

**Глава 10. Возврат в режим реальных адресов. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_in/chap_9.htm;j=false;s=1280*1024;d=24;rand=0.8397082243150186**

**Практически защищённый режим процессора можно использовать двумя способами:**

|  |  |
| --- | --- |
| **1.** | **Операционная система. Процессор входит в P-Mode при загрузке ОС и не возвращается в R-Mode. Примером таких ОС являются Windows 95 и старше либо специализированные ОС, управляющие контроллерами на базе 32-разрядных процессоров.** |
| **2.** | **DOS-программа. Программа запускается в ОС, которая работает в R-Mode (например, MS-DOS), переходит в P-Mode, выполняет свою работу, возвращается обратно в R-Mode и завершает свою работу. Примеров множество: это hymem.sys, обеспечивающий через P-Mode работу с памятью выше 1-го мегабайта (XMS-память); также это игрушки, работающие через Dos4GW - предзагружаемую ОС защищённого режима; это Windows 3.xx.** |

**Второй способ, с возвратом в режим реальных адресов, предоставляет программисту больше возможностей - ведь не все пишут полнофункциональные операционные системы...**

**Фактически, переход в режим реальных адресов может быть только из защищённого режима и осуществляется сбросом бита PE в CR0:**

**mov eax,cr0**

**and al,0feh**

**mov cr0,eax**

**или так**

**mov eax,cr0**

**btr eax,0**

**mov cr0,eax**

**Вы наверное уже догадались, что на самом деле не всё так просто. Действительно, выполнение трёх вышеприведенных команд переведёт процессор в R-Mode, но дальше он повиснет либо произойдёт аппаратный сброс, потому что программная среда не будет соответствовать режиму реальных адресов.**

**Для корректного перехода из P-Mode в R-Mode необходимо подготовить процессор следующим образом:**

|  |  |
| --- | --- |
| **1.** | **Запретить прерывания (CLI).** |
| **2.** | **Передать управление в читаемый сегмент кода, имеющий предел в 64Кб (FFFFh).** |
| **3.** | **Загрузить в SS, DS, ES, FS и GS селекторы дескрипторов, имеющих следующие параметры:**  ** Предел = 64 Кб (FFFFh)**  ** Байтная гранулярность (G = 0)**  ** Расширяется вверх (E = 0)**  ** Записываемый (W = 1)**  ** Присутствующий (P = 1)**  ** Базовый адрес = любое значение**  **Сегментные регистры должны быть загружены ненулевыми селекторами. Те сегментные регистры, в которые не будут загружены описанные выше значения, будут использоваться с атрибутами, установленными в защищённом режиме.** |
| **4.** | **Сбросить флаг PE в CR0.** |
| **5.** | **Выполнить команду far jmp на программу режима реальных адресов.** |
| **6.** | **Загрузить в регистры SS, DS, ES, FS и GS необходимые значения или 0.** |
| **7.** | **Разрешить прерывания (STI).** |

**В предыдущем примере (2) мы просто перевели процессор в защищённый режим, при этом не используя особенности прерываний в P-Mode и не подключали механизм виртуальной памяти. Реализация каждой из этих технологий немного усложнит возврат в R-Mode и в разделах, описывающих эти технологии будут приведены соответствующие примеры перехода в P-mode и возврата из него.**

**Здесь подразумевается, что вы уже ознакомились со всеми предыдущими главами, поэтому предварительного подробного описания перехода в R-Mode не будет и мы перейдём сразу к примеру.   
        Этот пример является полноценной программой и в нём без комментариев повторяется переход в P-Mode. Процесс подготовки дескрипторов и GDTR зависит от предназначения каждого примера и по этой причине я не вынес его в отдельную функцию а полностью описываю каждый раз.   
        Пример ориентирован на использование как самостоятельная .com-программа. Это сделано по следующим причинам:**

|  |  |
| --- | --- |
| **1.** | **Пользоваться обычной программной средой - библиотеки языков высокого уровня, прерывания MS-DOS и прочими особенностями этой ОС из защищённого режима вы не сможете.** |
| **2.** | **Размеры кода и данных в данном примере не большие и удобно использовать формат файла .com.** |
| **3.** | **.com-программу можно просто загрузить в память как блок данных и, передав управление на смещение 100h, использовать как оверлей.** |
| **4.** | **Вся мощь защищённого режима раскрывается при использовании 32-разрядного кода и данных. Инициализация защищённого режима должна происходить 16-разрядным кодом и 32-разрядные программы удобно использовать как отдельные модули.** |

**Также хочу обратить ваше внимание на то, что в этом примере адреса сегментов для защищённого режима совпадают с адресами сегментов, используемых в R-Mode. В результате, обращение к памяти происходит непосредственно через метки, определённые в исходнике (в предыдущем примере для этого требовалось из адреса метки вычитать адрес начала сегмента данных).**

**Пример используется вместе с файлом "pmode.lib".**

**Пример 3. Вход в защищённый режим и возврат в режим реальных адресов.**

**include pmode.lib**

**;---------------------------------------------------------------------------**

**.386p**

**pmode segment use16**

**assume cs:pmode, ds:pmode, es:pmode**

**org 100h**

**main proc far**

**start:**

**;------------------------------------------------------------------------**

**;------------------------------------------------------------------------**

**; Определяем селекторы как константы. У всех у них биты TI = 0 (выборка**

**; дескрипторов производится из GDT), RPL = 00B - уровень привилегий -**

**; нулевой.**

**Code\_selector = 8**

**Stack\_selector = 16**

**Data\_selector = 24**

**Screen\_selector = 32**

**R\_Mode\_Code = 40 ; Селектор дескриптора сегмента кода для возврата**

**; в режим реальных адресов.**

**R\_Mode\_Data = 48 ; Селектор дескриптора сегментов стека и данных.**

**;------------------------------------------------------------------------**

**; Сохраняем сегментные регистры, используемые в R-Mode:**

**mov R\_Mode\_SS,ss**

**mov R\_Mode\_DS,ds**

**mov R\_Mode\_ES,es**

**mov R\_Mode\_FS,fs**

**mov R\_Mode\_GS,gs**

**; Подготавливаем адрес возврата в R-Mode:**

**mov R\_Mode\_segment,cs**

**lea ax,R\_Mode\_entry**

**mov R\_Mode\_offset,ax**

**; Подготовка к переходу в защищённый режим:**

**mov bx,offset GDT + 8**

**xor eax,eax**

**mov edx,eax**

**push cs**

**pop ax**

**shl eax,4**

**mov dx,1024**

**mov cl,10011000b**

**call set\_descriptor ; Code**

**lea dx,Stack\_seg\_start**

**add eax,edx**

**mov dx,1024**

**mov cl,10010110b**

**call set\_descriptor ; Stack**

**xor eax,eax**

**mov ax,ds**

**shl eax,4**

**mov dx,0ffffh**

**mov cl,10010010b**

**call set\_descriptor ; Data**

**mov eax,0b8000h**

**mov edx,4000**

**mov cl,10010010b**

**call set\_descriptor ; Screen**

**; Готовим дополнительные дескрипторы для возврата в R-Mode:**

**xor eax,eax**

**push cs**

**pop ax**

**shl eax,4 ; EAX = физический адрес сегмента кода**

**; (и всех остальных сегментов, т.к.**

**; это .com-программа)**

**mov edx,0ffffh**

**mov cl,10011010b ; P=1, DPL=00b, S=1, Тип=101b, A=0**

**call set\_descriptor ; R\_Mode\_Code**

**mov cl,10010010b ; P=1, DPL=00b, S=1, Тип=001b, A=0**

**call set\_descriptor ; R\_Mode\_Data**

**; Устанавливаем GDTR:**

**xor eax,eax**

**mov edx,eax**

**mov ax,ds**

**shl eax,4**

**lea dx,GDT**

**add eax,edx**

**mov GDT\_adr,eax**

**mov dx,55 ; Предел GDT = 8 \* (1 + 6) - 1**

**mov GDT\_lim,dx**

**cli**

**lgdt GDTR**

**mov R\_Mode\_SP,sp ; Указатель на стек сохраняем в последний**

**; момент.**

**; Переходим в защищённый режим:**

**mov eax,cr0**

**or al,1**

**mov cr0,eax**

**; Процессор в защищённом режиме**

**db 0eah ; Команда far jmp Code\_selector:P\_Mode\_entry.**

**dw P\_Mode\_entry**

**dw Code\_selector**

**;------------------------------------------------------------------------**

**P\_Mode\_entry:**

**mov ax,Screen\_selector**

**mov es,ax**

**mov ax,Data\_selector**

**mov ds,ax**

**mov ax,Stack\_selector**

**mov ss,ax**

**mov sp,0**

**; Сообщаем о входе в P-Mode (выводим ZS-строку):**

**lea bx,Start\_P\_Mode\_ZS**

**mov di,480**

**call putzs**

**; Работа программы в защищённом режиме (здесь - только вывод строки):**

**lea bx,P\_Mode\_ZS**

**add di,160**

**call putzs**

**; Возвращаемся в режим реальных адресов.**

**; 1. Запретить прерывания (CLI).**

**; Прерывания уже запрещены при входе в P-Mode.**

**; 2. Передать управление в читаемый сегмент кода, имеющий предел в 64Кб.**

**db 0eah ; Команда far jmp R\_Mode\_Code:Pre\_R\_Mode\_entry.**

**dw Pre\_R\_Mode\_entry**

**dw R\_Mode\_Code**

**Pre\_R\_Mode\_entry:**

**; 3. Загрузить в SS, DS, ES, FS и GS селекторы дескрипторов, имеющих**

**; следующие параметры:**

**; 1) Предел = 64 Кб (FFFFh)**

**; 2) Байтная гранулярность (G = 0)**

**; 3) Расширяется вверх (E = 0)**

**; 4) Записываемый (W = 1)**

**; 5) Присутствующий (P = 1)**

**; 6) Базовый адрес = любое значение**

**mov ax,R\_Mode\_Data ; Селектор R\_Mode\_Data - "один на всех".**

**mov ss,ax**

**mov ds,ax**

**mov es,ax**

**mov fs,ax**

**mov gs,ax**

**; 4. Сбросить флаг PE в CR0.**

**mov eax,cr0**

**and al,0feh ; FEh = 1111'1110b**

**mov cr0,eax**

**; 5. Выполнить команду far jump на программу режима реальных адресов.**

**db 0eah**

**R\_Mode\_offset dw ? ; Значения R\_Mode\_offset и R\_Mode\_segment**

**R\_Mode\_segment dw ? ; сюда прописались перед входом в**

**; защищённый режим (в начале программы).**

**;------------------------------------------------------------------------**

**R\_Mode\_entry:**

**; 6. Загрузить в регистры SS, DS, ES, FS и GS необходимые значения или 0**

**; (восстанавливаем сохранённые значения):**

**mov ss,R\_Mode\_SS**

**mov ds,R\_Mode\_DS**

**mov es,R\_Mode\_ES**

**mov fs,R\_Mode\_FS**

**mov gs,R\_Mode\_GS**

**mov sp,R\_Mode\_SP ; Восстанавливаем указатель стека**

**; непосредственно перед разрешением**

**; прерываний.**

**; 7. Разрешить прерывания (STI).**

**sti**

**; Выводим ZS-строку "Back to real address mode..."**

**lea bx,R\_Mode\_ZS**

**mov ax,0b800h**

**mov es,ax**

**mov di,800**

**call putzs ; Функция putzs универсальна и работает**

**; в обоих режимах.**

**int 20h ; Конец программы (здесь - выход в MS-DOS).**

**;------------------------------------------------------------------------**

**; Вставка макросами кода функций, определённых в текстовом файле**

**; "pmode.lib":**

**init\_set\_descriptor**

**init\_putzs**

**;------------------------------------------------------------------------**

**; ZS-строка для вывода при входе в P-Mode:**

**Start\_P\_Mode\_ZS: db "Entering to protected mode...",0**

**; ZS-строка для вывода при работе в P-Mode:**

**P\_Mode\_ZS: db "Working in P-mode...",0**

**; ZS-строка для вывода в R-Mode:**

**R\_Mode\_ZS: db "Back to real address mode...",0**

**;------------------------------------------------------------------------**

**; Значения регистров, которые программа имела до перехода в P-Mode:**

**R\_Mode\_SP dw ?**

**R\_Mode\_SS dw ?**

**R\_Mode\_DS dw ?**

**R\_Mode\_ES dw ?**

**R\_Mode\_FS dw ?**

**R\_Mode\_GS dw ?**

**;------------------------------------------------------------------------**

**; Образ регистра GDTR:**

**GDTR label fword**

**GDT\_lim dw ?**

**GDT\_adr dd ?**

**;------------------------------------------------------------------------**

**GDT:**

**dd ?,? ; 0-й дескриптор**

**dd ?,? ; 1-й дескриптор (кода)**

**dd ?,? ; 2-й дескриптор (стека)**

**dd ?,? ; 3-й дескриптор (данных)**

**dd ?,? ; 4-й дескриптор (видеопамяти)**

**dd ?,? ; 5-й дескриптор (код для перехода в R-Mode)**

**dd ?,? ; 6-й дескриптор (стек и данные для перехода в R-Mode)**

**;------------------------------------------------------------------------**

**db 1024 dup (?) ; Зарезервировано для стека.**

**Stack\_seg\_start: ; Последняя метка программы - отсюда будет расти стек.**

**;------------------------------------------------------------------------**

**main endp**

**pmode ends**

**end start**