**Мультизадачность.**

Глава 1. Задачи. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_tss/main.htm;j=false;s=1280*1024;d=24;rand=0.5616630391638116

       Под термином задача (task) как правило подразумевается программа, выполняющая определённые действия, т.е. задача - это программа. Всякая программа характеризуется своей средой исполнения (execution environment) - это сегменты кода, которые она использует, сегменты и типы данных, порты ввода/вывода, различные устройства и даже состояние регистров, в общем, всё, с чем она взаимодействует.   
       Если компьютер управляется одной программой, то все ресурсы этого компьютера она может использовать монопольно и так, как ей это будет нужно, то есть в компьютере будет один "хозяин".   
       Если же необходимо выполнять параллельно несколько различных программ, то возникает ряд проблем, и все они связаны с тем, что компьютер один, а программ - много. Даже в мультипроцессорной системе, где каждой задаче можно выделить отдельный процессор, возникает проблемы с коллективным доступом к одиночным устройствам, например, шине данных, памяти, дисковой, сетевой, видео системам и пр.   
       То, что мультизадачность является важным свойством любой операционной системы, уже давно не подлежит сомнению. Сам процесс параллельной работы нескольких программ можно реализовать множеством способов, но только один из них будет наиболее надёжным - тот, где управление задачами будет происходить на аппаратном уровне. В 32-разрядных интеловских процессорах существует такой механизм, он достаточно мощный и гибкий, что значительно облегчает использование мультизадачности.   
       На самом деле любая мультизадачная система реализуется так, что одновременно выполняется только одна задача и лишь при быстром переключении с одной задачи на другую, создаётся иллюзия, что задачи работают параллельно. Так построены все операционные системы и иначе нельзя - ведь процессор может выполнять одновременно только одну программу, даже имея несколько параллельных конвейеров, всё равно он выполняет одну задачу. Каждая задача имеет свою среду исполнения и надёжность мультизадачной системы заключается в том, что на состояние любой задачи не может повлиять никакая другая. Пока работает текущая задача, все остальные находятся в состоянии паузы, в это время их команды не выполняются и данные не меняются.

       Для корректной работы мультизадачной системы нужны следующие основные факторы:

|  |  |
| --- | --- |
| 1. | Каждая задача (т.е. программа) должна быть изолирована от других, её код, данные и стек должны использоваться только ею самой и в идеале задача не должна "подразумевать" наличие других задач. |
| 2. | В мультизадачной системе должна быть одна задача - "хозяин", которая имеет высший приоритет по сравнению со всеми остальными задачами. Это нужно для того, чтобы переключать между собой задачи и управлять ресурсами, доступ к которым одновременно возможен для нескольких задач. |
| 3. | Использование разделяемых ресурсов (т.е. тех, которые разделяют между собой несколько задач, например, дисковая подсистема) должно быть корректным - либо каждая задача должна быть "в курсе" того, что кроме неё ещё кто-то может использовать этот ресурс, либо для каждого ресурса создаётся свой диспетчер - процедура или даже отдельная задача, которой принадлежит исключительное право на управление ресурсом. |
| 4. | При переключении с одной задачи на другую система должна обеспечить способ, благодаря которому состояние "старой" задачи не изменится состоянием "новой" задачи. |
| 5. | Необходимо учитывать временные характеристики - каждая задача должна подразумевать, что её выполнение может быть прервано в любой момент на неопределённое время. Этот фактор нужно учитывать при построении систем ввода/вывода, зависящих от времени, например, задача, управляющая модемом не должна терять данные по причине своего прерывания. |

       В 32-разрядных процессорах предоставляется только базовый аппаратный механизм управления мультизадачностью, структура же самих задач, их работа и взаимодействия друг с другом целиком ложатся на плечи программиста и в связи с этим возникает множество возможных вариантов построения мультизадачной системы. Нельзя сказать, что какой-то метод самый лучший - всё зависит, во-первых, от предназначения ОС, а во-вторых, и это основная причина, от времени, которое вы можете затратить на построение ОС - нормальным явлением будет многократная переделка и усовершенствование структуры задач и их управления.   
       В этом разделе вашему вниманию будут представлены несколько примеров мультизадачных систем, сложность которых будет постепенно возрастать. Эти примеры приводятся только для того, чтобы показать, **как именно** реализуется мультизадачность, но ни в коем случае не рассматривайте эти примеры, как единственно или наиболее правильные варианты построения мультизадачности, т.к. эта часть ОС сильно зависит от её предназначения и является одной из главных характеристик любой системы.

Глава 2. Локальные дескрипторные таблицы. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_tss/chap_1.htm;j=false;s=1280*1024;d=24;rand=0.41415822819571624

        Локальная дескрипторная таблица LDT (Local Descriptor Table) является аналогом глобальной дескрипторной таблицы GDT и предназначена для применения **в контексте задачи**. Таким образом, задача может иметь свой собственный набор дескрипторов, непосредственный доступ к которым имеет только она сама; операционная система также может использовать локальные дескрипторы задачи, но явного механизма для этого в процессоре не предусмотрено и это зависит от конкретной системы. Благодаря локальной дескрипторной таблице, которые мы далее будем называть просто LDT, и грамотному построению ядра ОС можно гарантировать, что кроме данной задачи никакая другая не сможет воспользоваться её дескрипторами, что, несомненно, повышает защищённость системы.   
        LDT имеет точно такую же структуру, как и GDT; в ней так же нельзя использовать нулевой дескриптор и всё же LDT имеет ряд отличий от GDT:

|  |  |
| --- | --- |
| 1. | В LDT нельзя использовать дескрипторы некоторых системных объектов, подробнее об этом вы узнаете далее. |
| 2. | LDT можно использовать только в контексте задачи. Для использования задач необходима инициализация некоторых дескрипторов и структур в памяти, которые собственно и определяют контекст задачи, поэтому просто так, в обычной процедуре, подобной тем, что мы до сих пор использовали, LDT применить нельзя. |
| 3. | Одновременно в задаче можно использовать только одну LDT, зато их можно менять на другие, т.е. переключаться с одной LDT на другую, однако это только теоретическое использование LDT, реально необходимости в её смене не возникает. |

        Благодаря наличию LDT программа может использовать одновременно уже две дескрипторных таблицы - GDT и LDT. Дескрипторы из LDT выбираются селекторами, так же, как и из GDT, но тут необходимо каждый раз явно указывать, откуда именно происходит выборка - из LDT или GDT.   
        Давайте вспомним формат селектора:

http://sasm.narod.ru/docs/pm/pm_img/pm_in5_1.gif  
**Рис. 2-1. Формат селектора.**

        Обратите внимание на 2-й бит селектора. Он называется TI (Table Indicator) и определяет, к какой дескрипторной таблице производится обращение - если TI = 0, то к GDT, если TI = 1 - к LDT.   
        До сих пор мы использовали только нулевое значение этого бита, кроме того, поле RPL селектора мы также везде устанавливали нулевым, отсюда и получались значения селекторов, кратные восьми: 8 - для кода, 16 - для стека, 24 - для основного сегмента данных и т.д.; с использованием задач нам станут доступны все возможные значения селекторов.

        Итак, селектор, имеющий одинаковый индекс, позволяет выбирать дескриптор как из GDT, так из LDT. Например, для выборки первого дескриптора из таблицы (нулевой дескриптор не используется!) нужно, чтобы поле индекса селектора содержало единицу (1). Если предположить, что поле PRL будет нулевым (как мы до сих пор и делали), то для выборки первого дескриптора:

|  |  |
| --- | --- |
|  | из GDT, нам понадобится селектор со значением 1000b; |
|  | а из LDT - селектор со значением 1100b. |

        Обратите внимание, что бит TI определяет из какой именно таблицы производится выборка, а поле индекса определяет, какой именно дескриптор будет выбран.

        Практически, при обращении к LDT (опять же подразумевая, что поле RPL в селекторе будет содержать 0), можно определить селекторы, например, так:

Для GDT:

Code\_Selector = 8

Data\_Selector = 16

Stack\_Selector = 24

Для LDT:

LDT\_Code\_Selector = 8 + 100b

LDT\_Data\_Selector = 16 + 100b

LDT\_Stack\_Selector = 24 + 100b

        Локальная дескрипторная таблица представляет собой по сути массив 8-байтных дескрипторов, однако, для использования LDT нужно определить специальный дескриптор системного объекта - дескриптор LDT. Этот дескриптор имеет следующий формат:

dw limit\_low ; Младшая часть предела

dw address\_low ; Младшая часть базового адреса

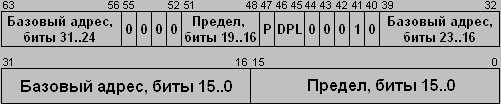
db address\_mid ; Третий (из четырёх) байт базового адреса

db access\_rights ; Права доступа

db limit\_hi ; Старшая часть предела

db address\_mid ; Четвёртый (старший) байт базового адреса

        Более наглядно это показано на рисунке 2-1:

  
**Рисунок 2-1. Формат дескриптора LDT**

        Как видите, формат дескриптора LDT очень похож на формат обычного дескриптора сегмента данных или кода, за исключением содержимого байта прав доступа и нулей вместо бит GDXU (старшие 4 бита в байте со старшей частью предела) - здесь эти биты зарезервированы и должны быть равными нулю.

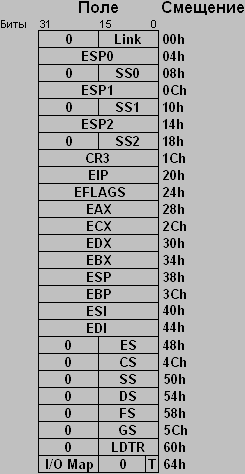
        Для установки дескрипторов в LDT и их динамического изменения, обращаться к ней через дескриптор LDT нельзя. Дескриптор LDT - это **системный** объект, он предназначен для автоматического использования процессором (в этом и состоит отличие системных объектов от дескрипторов сегментов; мы уже рассмативали системные объекты - это были шлюзы, из которых строится дескрипторная таблица прерываний IDT). Поэтому для работы с LDT любой задачи нужно определять алиасы (от англ. alias) - в данном случае под алиасом подразумевается сегмент данных, отображённый на те же адреса, по которым располагается LDT задачи.   
        Точно также для динамического использования GDT нужно определять алиасы - как правило определяют отдельный дескриптор сегмента данных, имеющий такой же адрес и предел, что и GDT.

        В мультизадачной системе подразумевается наличие нескольких задач и, следовательно, нескольких LDT, поэтому удобно эти LDT размещать последовательно в памяти, чтобы их можно было описать одним алиасным дескриптором данных. В наших примерах мы будем определять отдельный дескриптор сегмента данных LDT\_area, в котором мы будет размещать таблицы LDT всех задач. При конструировании самих дескрипторов LDT мы будем вычислять их адреса, используя базовый адрес сегмента LDT\_area.

        Использование LDT в задаче не является обязательным. Если все задачи работают на нулевом уровне привилегий, то нет необходимости определять для них отдельные дескрипторы в LDT - достаточно использования GDT. Если задача с PL=0 является сама по себе достаточно сложной, например, менеджер дисковой подсистемы, тогда для удобства программиста можно определить LDT с набором дескрипторов для этой задачи.   
        Полностью эффективность использования LDT открывается при использовании задач, работающих на не нулевых уровнях. Например, прикладной задаче с PL=3 система может определить LDT с необходимым набором дескрипторов и таким образом изолировать её на третьем уровне привилегий.

Глава 3. Сегмент состояния задачи. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_tss/chap_2.htm;j=false;s=1280*1024;d=24;rand=0.4221811011601564

        Сегмент состояния задачи TSS (Task State Segment) - это структура данных, которая определяет состояние (т.е. контекст) задачи. В ней хранится содержимое всех регистров общего назначения, сегментных и некоторых системных регистров а также некоторая дополнительная информация.   
        Для каждой задачи в системе нужно определить один сегмент TSS и далее процессор будет сам использовать эти сегменты задач, т.е. структура TSS поддерживается аппаратно.   
        TSS бывают двух типов - 16- и 32-разрядные. 16-разрядные появились в процессоре i286 и они могут использоваться в 32-разрядных процессорах, но мы их рассматривать не будем, потому что программирование Intel286 здесь не рассматривается.   
        32-разрядные TSS появились в процессоре Intel386, с тех пор они не претерпели изменений и на них строится механизм мультизадачности любой ОС защищённого режима. Везде далее, где мы будем говорить о TSS, именно такие, 32-разрядные, и будут подразумеваться.   
        На рис. 3-1 приведена структура 32-разрядного TSS:

  
**Рисунок 3-1. Структура TSS**

        Как видите, большая часть полей предназначена для хранения содержимого регистров общего назначения и сегментных регистров. Остальные поля:

|  |  |
| --- | --- |
|  | **Link** (смещение 0) - это поле обратной связи. Когда процессор переключается с одной задачи на другую, он записывает в это поле информацию о предыдущей задаче, чтобы "знать", куда ему возвращать управление после того, как эта задача отработает. Подробнее об этом поле см. далее. |
|  | **SSi:ESPi** (i = 0,1,2) - это пары значений указателя на стек для разных уровней привилегий - 0, 1 и 2. Задачу можно вызывать с разных уровней привилегий и для корректной работы нужно разделение стека. Работа с этими полями - отдельная тема и пока мы использовать их не будем. |
|  | **CR3** (смещение 1Ch) - это содержимое регистра CR3 для данной задачи. Как вы помните, CR3, он же PDBR, определяет параметры каталога страниц. То, что каждая задача имеет в своём TSS поле CR3 означает, что каждая задача может иметь свой "личный" набор страниц и работать по своей собственной схеме страничного преобразования. Однако, на практике, удобнее использовать одно значение для всех CR3 - это повышает надёжность и защищённость системы. |
|  | **LDTR** (смещение 60h) - это селектор дескриптора LDT, который используется данной задачей. Если задаче не нужна LDT, то в этом поле хранится 0 - вот здесь проявляется смысл условия, по которому нулевой дескриптор в GDT не используется - нулевой селектор означает отсутствие этого селектора. Если LDT используется, то задача может иметь свой собственный набор дескрипторов. |
|  | **T** (0-й бит по смещению 64h) - флаг трассировки, применяется для отладки задач, процессор его только считывает. Если этот флаг установлен, то при переключении на данную задачу процессор генерирует исключение отладки (прерывание 1). Этот флаг также может быть полезен в системах, использующих FPU (сопроцессор), MMX и XMM. Как вы заметили, в TSS нет полей для значений регистров этих модулей, поэтому для сохранения и загрузки контекста этих устройств требуется дополнительные усилия со стороны операционной системы. |
|  | **I/O Map** (смещение 66h) - это адрес карты ввода вывода, точнее - смещение её начала от начала TSS. Каждая задача может иметь такую карту, в которой каждый бит определяет возможность доступа к конкретному порту - установлен - нет доступа, сброшен - есть. Таким образом реализуется защита на уровне доступа к портам ввода/вывода. Подробнее об этом - в главе 5. |
|  | **Дополнительная часть TSS** (начиная со смещения 68h) - это та часть TSS, которая предусматривается программистом при создании данной задачи. Здесь должна храниться карта ввода/вывода (если она есть), здесь также может быть контекст FPU, MMX и XMM и другая информация, в зависимости от конкретной реализации ОС. Процессор непосредственно обращается только к карте ввода/вывода, к остальной информации он явно не обращается и на размер этой части не накладывается особых ограничений (т.е. не более 1Мб). |

        TSS определяется специальным дескриптором. Такой дескриптор является системным объектом и может находиться только в GDT (в LDT - нельзя), а это значит, что задача самостоятельно не может определять для себя другие задачи в LDT и для операционной системы есть условия для контроля задач.   
        Формат дескриптора TSS следующий:

dw limit\_low ; Младшая часть предела

dw address\_low ; Младшая часть адреса

db address\_mid ; 3-й (из 4-х) байт адреса

db access\_rights ; Права доступа

db limit\_hi ; Старшая часть предела

db address\_hi ; Старший байт адреса

Примечание 1:   
        Байт прав доступа access\_rights имеет следующий формат:

бит: описание

0: = 1

1: Бит B (Busy) - занятость задачи (см. следующую главу)

2: = 1

3: = 0

4: = 0

5,6: = DPL - Уровень привилегий сегмента TSS (см. следующую главу)

7: P - бит присутствия сегмента, обычно установлен в 1.

Примечание 2:   
        Старшая часть предела limit\_hi имеет такой формат:

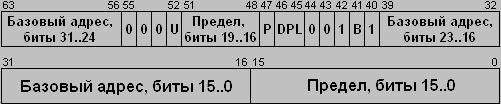
бит: описание

0..3: Старшие 4 бита предела

4: Бит U (User) - этот бит ОС может использовать в своих целях

5..7: = 0

        Для наглядности формат дескриптора TSS приведен на рис. 3-1:

  
**Рисунок 3-2. Формат дескриптора TSS**

        Дескриптор TSS определяет сегмент состояния задачи. Этот дескриптор описывает системный объект, т.к. бит S в нём равен 0 (это 44-й бит дескриптора или 4-й бит прав доступа). То, что дескриптор описывает системный объект означает, что непосредственно пользоваться этим объектом может только процессор, но не программа - любая попытка загрузки селектора этого дескриптора в сегментный регистр вызовет генерацию исключения общей защиты (#GP - прерывание 0Dh).   
        Как уже говорилось в предыдущей главе, в локальной дескрипторной таблице LDT нельзя использовать некоторые системные объекты. Т.к. дескрипторы LDT и TSS являются такими объектами, то их нельзя применять в LDT, иначе при обращении к ним будет генерироваться исключения. Смысл такого ограничения состоит в том, что задаче запрещено определять задачи внутри себя и устанавливать дополнительные дескрипторные таблицы и таким образом, эти действия может выполнять только ядро ОС.   
        Обратите внимание на то, что в сегменте состояния задачи нет полей для регистров модулей FPU, MMX и XMM. Дело в том, что не всякая задача работает с каким-либо из этих модулей и наличие дополнительных полей замедлило бы переключение задачи. Сегмент TSS и так достаточно большой, к тому же поля из него загружаются по одному - процессор выполняет проверку параметров значений каждого сегментного и системного регистра.   
        Если в системе несколько задач использует FPU, MMX и/или XMM, то при смене задач необходимо менять контекст соответствующего модуля. Процессор автоматически этого не делает, зато он создаёт условия для операционной системы, позволяя ей дополнить контекст задачи, например, сохранить состояние FPU, используемое старой задачей и загрузить в FPU значения, используемые новой задачей.

Глава 4. Управление задачами. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_tss/chap_3.htm;j=false;s=1280*1024;d=24;rand=0.6502364803991634

        Переключение на задачу процессор производит автоматически и он сам использует дескриптор TSS и все поля из которых состоит сегмент состояния задачи.   
        Переключение производится командами далёкой передачи управления:

|  |  |
| --- | --- |
|  | far jmp |
|  | far call |
|  | iret |

        Каждая из этих команд имеет адрес перехода, состоящий из двух частей - селектора сегмента и смещения. Для того, чтобы переключиться на задачу, нужно в качестве селектора указать селектор TSS, смещение при этом будет проигнорировано процессором, потому что в самом TSS описаны все регистры задачи, в частности, CS и EIP - вот они и определят, откуда именно будет выполняться задача.   
        Для переключения на задачу в процессоре не предусмотрено специальных команд, просто когда процессор обнаруживает, что селектор в адресе далёкого перехода указывает на дескриптор TSS, то запускается механизм переключения задач.

        Используя задачи в своей операционной системе, вам необходимо их рассматривать как отдельные процессы, точнее - как самостоятельные программы, но ни в коем случае не как альтернативу процедурам. В принципе, можно построить всю ОС, используя для каждой сложной процедуры отдельную задачу, но такая ОС не будет эффективна. Во-первых, передача управления в далёкую процедуру с возвратом обратно занимает в 20 раз меньше времени, чем переключение на задачу и обратно. Это связано с тем, что при переключении на задачу происходит загрузка сегментных и системных регистров и для каждого процессор выполняет проверку параметров соответствующего дескриптора. Во-вторых, задачи и механизм их поддержки в процессоре предназначены для того, чтобы надёжно изолировать одну программу от остальных и реализовать настоящую мультизадачность.   
        Вспомните, как реализовывалась мультизадачность в режиме реальных адресов, точнее - в MS-DOS: благодаря множеству резидентных программ, "висящих" на различных прерываниях, реализуется "параллельная" работа нескольких программ, но на самом деле, весь MS-DOS со всеми его программами и резидентами можно рассматривать как одну большую программу, потому что разные "задачи" в ней не разделены между собой и могут запросто взаимодействовать с кодом и данными любой другой "задачи".   
        В защищённом режиме такого нет. Процессор попросту не допустит любой задаче выйти за поставленные ей рамки, которые и определяются содержимым TSS - его дескрипторами, LDT, CR3 и картой разрешения ввода/вывода. Любая попытка нарушить условия защиты приведёт к генерации исключения и операционная система будет "знать" о нарушителе всё. Что дальше с ним делать уже определяется идеологией самой ОС - можно безжалостно уничтожать задачи-нарушители, можно игнорировать нарушения, но не допускать их, а можно эмулировать выполнение запрещённых действий и "шпионить" за подозрительными задачами - например, вычислять вирусы.

        Для того, чтобы переключиться с одной задачи на другую, нужно чтобы процессор уже выполнял задачу, т.е. переключиться на задачу можно только из задачи. Это значит, что, во-первых, просто запустить первую задачу не получится, нужны некоторые хитрости, о которых мы поговорим позже. А во-вторых, из задачи нельзя "вернуться" в обычную процедуру, т.е. однажды запустив мультизадачность, программа остаётся в ней до перехода в режим реальных адресов.   
        Переключение задач происходит следующим образом. Процессор сначала сохраняет параметры текущей задачи в её TSS, заполняя все поля, начиная с CR3. Потом он ищет в GDT дескриптор указанной новой задачи и после проверки всех параметров дескриптора, начинает загружать регистры значениями из полей нового TSS. При этом процессор контролирует значение каждого системного и сегментного регистра и если обнаруживает ошибку, то генерирует исключение. После загрузки всех значений, процессор записывает в поле Link (смещение 0 в TSS) селектор дескриптора предыдущей задачи и немного корректирует значение EFLAGS. После всего этого процессор передаёт управление по адресу CS:EIP и задача начинает свою работу (или продолжает, если она была прервана).   
        Как видите, переключение задачи довольно-таки трудоёмкий процесс и требует много времени на различные проверки и обращения к памяти. Кроме того, переключение на задачу не кэшируется буфером ветвлений, представленных в процессорах, начиная с Pentium и при переключении на задачу происходит сброс конвейера команд, что потребует перед выполнением первой команды задачи потерю времени на ожидание, пока заполнится линия кэша из памяти и пока команды пройдут по конвейеру. Дополнительные задержки происходят из-за того, что при чтении новых значений из TSS, происходит загрузка CR3, а это значит, что производится сброс буферов TLB и процессору понадобится обращаться к каталогу и таблицам страниц, даже если он только что с ними работал (загрузка значения в CR3, даже того же самого, что в нём и было, приводит к сбросу буферов TLB).

        Наблюдая столько сложностей и проблем с этими задачами, возникает вполне логичный вопрос о целесообразности их применения, однако задачи процессором **поддерживаются аппаратно** и только из-за этого уже стоит их применять. К тому же, с каждым новым процессором Intel увеличивает производительность в первую очередь в таких проблемных местах, как переключение задач и в современных процессорах оно происходит очень быстро.   
        Из-за того, что переключение задач происходит относительно медленно и приводит к полному сбросу некоторых кэшей, переключение задач нужно использовать как можно реже и не применять отдельной задачи для одной процедуры. Например, вывод пикселя на экран не стоит делать одной задачей. То же самое относится к выводу символа или любого другого графического объекта, т.е. использование задач как альтернатива прерываниям MS-DOS (в данном случае - каждому сервису - отдельная задача) так же приведёт к неэффективному использованию мультизадачности. Такие вещи, как сервисы ОС, лучше выполнять в виде процедур или, в крайнем случае - прерываний, а задачами реализовывать прикладные программы и базовые модули ОС (например, менеджер дисковой подсистемы).

        Передача управления от задачи к задаче может происходить тремя способами:

|  |  |
| --- | --- |
| 1. | Одна задача вызывает другую командой FAR CALL, а вторая задача возвращает управление первой командой IRET. |
| 2. | Задача переключается на другую задачу командой FAR JMP. |
| 3. | Происходит прерывание - либо аппаратное (IRQ), либо исключение и обработчик такого прерывания является отдельной задачей. Обработчик возвращает управление прерванной задаче командой IRET. |

        Итак, команда IRET может производить переключение задач. Как вам известно, обычное назначение этой команды - возврат из прерывания, т.е. она отличается от RET тем, что извлекает из стека ещё и регистр флагов (FLAGS, если стек - 16-разрядный, EFLAGS - если 32-разрядный).   
        Как определить, что же именно выполнит данная команда - переключение на задачу или возврат из прерывания? Для этого в регистре флагов EFLAGS введён специальный флаг NT (Nested Task - вложенная задача). Каждый раз, когда процессор выполняет команду FAR CALL, вызывая задачу, этот флаг устанавливается в 1. При выполнении команды IRET процессор проверяет этот флаг - если он равен 1, то он переключается на предыдущую задачу (селектор которой хранится в поле Link по смещению 0 в TSS текущей задачи). Если NT = 0, происходит возврат из прерывания.

        Если переключение на задачу было произведено командой FAR CALL либо оно произошло в результате прерывания, то процессор после загрузки контекста новой задачи установит флаг NT в регистре EFLAGS. Это обеспечит возврат в старую задачу командой IRET.

        Обратите внимание на бит B (Busy) в дескрипторе TSS. Этот бит определяет занятость задачи. Каждый раз, когда процессор переключается на задачу, он устанавливает этот бит. Когда задача передаёт управление другой задаче командой FAR JMP или IRET, процессор сбрасывает флаг B в дескрипторе текущей задачи (той, из которой происходит передача управления).   
        Передача управления командой FAR CALL не сбрасывает флаг B. Если происходит прерывание, то этот флаг в прерванной задаче также не сбрасывается. Таким образом, установленный бит B показывает, что данная задача занята, т.е. она либо работает в данный момент, либо она прервана (находится в состоянии паузы).   
        Необходимость бита занятости B заключается в том, что процессор запрещает передачу управления занятой задаче (он генерирует исключение общей защиты). Это сделано для того, чтобы задачи не могли себя вызывать рекурсивно. Вот здесь проявляется отличие задачи от процедуры - задача, как программа, рекурсивной не бывает.   
        Однако, при помощи манипуляций непосредственно над содержимым дескрипторов TSS (сброс и установка флага B "вручную") можно добиться переключения на занятую задачу. Это иногда бывает необходимым, в частности, при запуске мультизадачности.

        Дескриптор TSS содержит в байте прав доступа поле DPL (Descriptor Privilege Level). В данном случае, для дескриптора TSS, это поле содержит уровень привилегий, **с которого** процессор разрешает переключение на эту задачу. Переключения с меньшего уровня привилегий также разрешены. Другими словами, при передаче управления должно выполняться следующее условие:

CPL <= DPL,

где CPL - это Current Privilege Level - текущий уровень привилегий (т.е. тот, на котором выполняется код в данный момент).   
        Это ещё одно средство контроля целостности системы. Для системных задач поле DPL нужно устанавливать в 00b, для прикладных задач - в 11b, при этом системная задача, с DPL = 00b может переключаться на задачу с любым DPL, а, например, задача с DPL = 10b - на задачи с DPL = 10b и 11b.

Глава 5. Карта разрешения ввода/вывода. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_tss/chap_4.htm;j=false;s=1280*1024;d=24;rand=0.6751237312801529

        Благодаря применению мультизадачности, операционная система может контролировать операции ввода/вывода любой задачи на уровне отдельных портов. Этот контроль производится аппаратно и в случае попытки доступа к запрещённым портам ввода/вывода генерируется исключение общей защиты.   
        Каждая задача может иметь карту разрешения ввода/вывода. Эта карта должна располагаться в TSS задачи и её параметры задаётся следующим образом:

|  |  |
| --- | --- |
|  | в TSS в dw-поле "I/O Map" (по смещению 66h) находится базовый адрес карты ввода/вывода, этот адрес - смещение начала карты от начала TSS; |
|  | сверху карта ограничена пределом TSS, установленным в дескрипторе TSS. |

        Каждый бит карты соответствует одному однобайтовому порту. Например:

|  |  |
| --- | --- |
|  | при выводе **байта** в порт по адресу 21, вывод будет в этот порт и будет проверяться 21-й бит карты - это 5-й (от нуля) бит в 2-м (от нуля) байте этой карты. |
|  | при выводе **слова** в порт 21, вывод будет производится в порты 21 и 22 и поэтому проверяются 2 бита из карты - 21-й и 22-й. |
|  | при выводе **двойного слова** в порт 21, будет происходить вывод в порты 21, 22, 23 и 24, и проверяться в карте будут биты с 21 по 24. |

        Если бит в карте разрешения ввода/вывода сброшен, то вывод разрешён, если установлен - то запрещён. Таким образом, если при операции ввода/вывода двойного слова процессор обнаружит, что хотя бы один из 4-х бит в карте установлен, он сгенерирует исключение общей защиты - обращение к порту запрещено.

        В регистре флагов EFLAGS есть двухбитовое поле IOPL - I/O Privilege Level (уровень привилегий для ввода/вывода). Это поле определяет максимальный уровень привилегий, который может иметь код задачи при операциях ввода/вывода. До сих пор мы везде сбрасывали в 00b это поле - это делалось неявно, при помощи записи 0 в регистр EFLAGS сразу после перехода в защищённый режим. Мы пока с портами не работаем, следовательно, поле IOPL не используем и здесь о нём говорится лишь для иллюстрации работы карты разрешения ввода/вывода.

        Карта разрешения ввода/вывода используется, когда процессор работает в режиме виртуального 8086, либо когда в задаче CPL > IOPL. Если процессор находится в защищённом режиме и выполняет задачу, CPL которой не больше её IOPL, то операции ввода/вывода разрешены по всем адресам.   
        Таким образом, обычно для всех системных задач ввод/вывод разрешается по всем адресам, а для задач на ненулевых уровнях привилегий вывод либо запрещается вообще, либо разрешается по определённым адресам. Например, драйвера устройств в ОС расположены на 1-м уровне привилегий и имеют карту ввода/вывода с разрешёнными портами для конкретного устройства или класса устройств, обслуживаемых этим драйвером, а прикладным программам, расположенным на 3-м уровне привилегий, ввод/вывод вообще запрещён. Благодаря этому, все операции с портами производятся через драйвера, надёжность которых всегда подразумевается выше, чем прикладные программы и такая система получается устойчивой к несанкционированному доступу к портам и, следовательно, к устройствам.

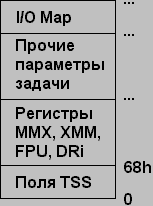
        Карта разрешения ввода/вывода может располагаться по любому смещению в TSS и иметь любой размер. Если адрес карты больше, чем предел TSS, либо равен ему, то считается, что карты нет и все команды ввода/вывода (при проверке карты) будут генерировать исключение. Адрес карты должен быть не более DFFFh.

        Каждый раз при обращении к карте ввода/вывода, процессор считывает два байта - тот, в котором начинается адрес порта и следующий, поэтому, карта должна быть на 1 байт больше, чем её размер и этот последний байт должен быть равен 1111'1111b. Например, если карта описывает первые 80 портов с адресами от 0 до 79, то такая карта должна состоять из 10 байт, а сразу за последним байтом карты должен быть байт со значением FFh. Предел карты в таком случае будет равен 10.

        Карта ввода/вывода должна размещаться в конце TSS, т.к. предел TSS задаёт также и конец карты. Карта описывает порты с самого начала, т.е. с адреса 0. Всё адресное пространство портов ввода/вывода в интеловских процессорах составляет 65'536 портов, т.е. 64 Кпорта, для описания всего адресного пространства понадобится 64 Кбита или же 8Кб. Таким образом, если вы реализуете задачей драйвер видеокарты, который переключает видеоокна через порт 3d8h, то карта ввода/вывода должна состоять минимум из 3d9h бит, т.е. из 125 байт (последний будет равен FFh).

        При конструировании дескриптора TSS необходимо всегда помнить, что минимально допустимый предел TSS для любой задачи равен 67h. TSS с таким пределом имеет размер в 68h, т.е. в 104 байт и содержит все поля. Если при переключении на задачу процессор обнаружит, что в дескрипторе TSS указан меньший предел, то он сгенерирует исключение недопустимого TSS (#TS).

        Наиболее общим видом TSS можно предположить следующий:



        Это, разумеется, не единственный возможный вид TSS, но главное то, что здесь схематически показано расположение полей TSS и карты разрешения ввода/вывода.

Глава 6. Регистр задачи TR. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_tss/chap_5.htm;j=false;s=1280*1024;d=24;rand=0.5637123958684076

        TR (Task Register) - это 16-разрядный системный регистр, который хранит селектор дескриптора TSS текущей задачи. Этот регистр также имеет теневую 64-разрядную компоненту, используемую только самим процессором, в которой хранится содержимое дескриптора TSS текущей задачи - это повышает производительность процессора.   
        Когда происходит переключение со старой задачи на новую, процессор помещает в TSS новой задачи в поле Link содержимое регистра TR и таким образом обеспечивает связь в со старой задачей. После загрузки значений из TSS новой задачи, в регистр TR процессор записывает селектор дескриптора TSS этой новой задачи.   
        Для загрузки значения в регистр TR используется команда LTR, единственным операндом которой служит 16-разрядный регистр общего назначения или переменная в памяти. Для чтения значения из этого регистра используется команда STR, которая также имеет один операнд - 16-разрядный регистр общего назначения или переменную в памяти.   
        Команда LTR относится к привелегированным командам - она может выполняться только на нулевом уровне привилегий. Команду STR можно выполнить на любом уровне привилегий, так что любая программа или задача может прочитать значение этого регистра. Это может показаться странным - процессор позволяет прикладным задачам считывать "конфиденциальную" информацию - селектор дескриптора TSS, однако, на самом деле, информации значение TR программе не даёт, потому что в этом регистре находится селектор дескриптора TSS текущей задачи и задача ничего с этим сделать не сможет - переключения на саму себя запрещены.

        Регистр TR имеет, в основном, два применения:

|  |  |
| --- | --- |
| 1. | Считывая значение TR, программа может определить текущую выполняемую задачу. Это может пригодится при отладке, например, для вывода на экран селектора дескриптора TSS текущей задачи, либо, для тех обработчиков прерываний и исключений, которые не являются сами задачами и выполняются в контексте текущей задачи, значение из TR позволяет определить текущую задачу. |
| 2. | Для перевода процессора в режим мультизадачности необходима загрузка селектор в регистр TR. Как именно это делается, вы увидите на примерах. |

        При загрузке регистра TR в дескрипторе TSS задачи устанавливается флаг занятости B. Загрузка этого регистра связывает текущее состояние процессора с контекстом данной задачи. Для перевода процессора в режим мультизадачности, достаточно выполнить следующие действия:

; Main\_TSS - это селектор некоторого определённого дескриптора TSS.

mov ax,Main\_TSS

ltr ax ; Загружаем в регистр TR селектор

; дескриптора TSS задачи Main. Теперь текущая

; задача - это Main\_TSS.

; Чистим Busy flag Main\_TSS - он установился после загрузки селектора в TR

mov bx,offset GDT + Main\_TSS

and byte ptr [ bx + 5 ],11111101b

; Переход на Main\_TSS

db 0eah ; Этот код эквивалентен команде FAR JMP Main\_TSS:00

dw 00

dw Main\_TSS

; Вот здесь процессор сохранил контекст текущей задачи, т.е. заполнил

; поля TSS текущими значениями.

; Main\_TSS:

; А теперь процессор загрузил те же самые значения из TSS в регистры.

; Теперь процессор в режиме мультизадачности.

; Установка Busy flag Main\_TSS - она нужна, потому что был переход с

; Main\_TSS на Main\_TSS командой JMP - эта команда сбрасывает флаг занятости

; старой задачи, т.е. Main\_TSS.

mov bx,offset GDT + Main\_TSS

or byte ptr [ bx + 5 ],10b

Глава 7. Задачи и флаги. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_tss/chap_6.htm;j=false;s=1280*1024;d=24;rand=0.6526711129818686

        Для управления задачами процессор использует несколько флагов:

|  |  |
| --- | --- |
|  | **Флаг занятости B (Busy)** - находится в дескрипторе TSS, в байте прав доступа. Устанавливается всякий раз, когда происходит переключение на задачу и когда установлен, означает, что задача занята. Переключение на занятую задачу запрещено и этот флаг предназначен для предотвращения рекурсивного вызова задачи. Флаг занятости сбрасывается при переключении на другую задачу командами FAR JMP либо IRET; при переключении командой FAR CALL либо при прерывании (даже если обработчик прерывания - тоже задача) флаг не сбрасывается. |
|  | **Флаг трассировки T** - находится в сегменте состояния задачи TSS, это 0-й бит по смещению 64h в TSS. Если флаг установлен, то при переключении на задачу процессор сначала загрузит значения из всех полей TSS, проверит их правильность и, если не обнаружит нарушений, сгенерирует исключение отладки (прерывание 1). Если флаг сброшен, то при переключении на задачу исключение отладки не генерируется. Этот флаг предназначен для отладки задач и также может применяться для явного системного дополнения контекста задачи, например, для загрузки регистров FPU. |
|  | **Флаг NT (Nested Task)** - находится в регистре EFLAGS. Если переключение на новую задачу было вызвано командой FAR CALL либо старая задача была прервана исключением или прерыванием и его обработчик также является задачей, то флаг NT устанавливается в регистре EFLAGS новой задачи. Благодаря этому новая задача может вернуть управление старой задаче командой IRET. Команда IRET выполняет одно из двух действий:   |  |  | | --- | --- | |  | Если NT = 0, то производит обычный возврат из прерывания; | |  | Если NT = 1, то производит переключение на предыдущую задачу, селектор дескриптора TSS которой находится в поле Link в TSS текущей задачи. | |
|  | **Флаг TS (Task Switched)** - находится в регистре управления CR0. Этот флаг устанавливается каждый раз, когда процессор переключается на задачу и служит индикатором переключения задач. При попытке выполнить команды FPU, MMX или XMM, процессор может генерировать исключение неприсутствующего устройства (#NM - прерывание 7), что позволяет системе выполнить смену контекста FPU, MMX и XMM. |

        Процессор позволяет пользоваться режимом виртуального 8086 только в контексте задачи и этот режим добавляет некоторые другие флаги, с которыми взаимодействует процессор в задаче, но такой режим процессора довольно сложный и пока мы его рассматривать не будем.   
        Взаимодействия задач с флагами приведено в таблице 7-1:

**Таблица 7-1.** Воздействие при переключении задачи на флаги Busy, NT, TS и поле Link.

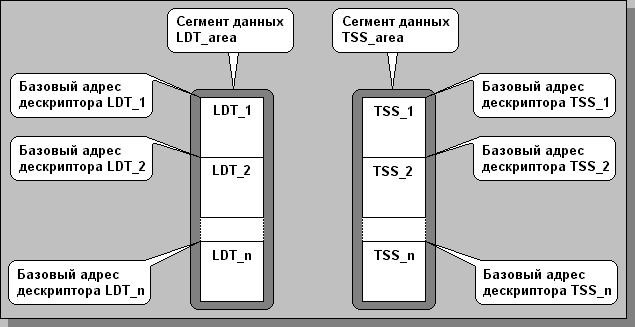
|  |  |  |  |
| --- | --- | --- | --- |
| Флаг или поле | Эффект от команды JMP | Эффект от команды CALL или прерывания | Эффект от команды IRET |
| Флаг B (Busy) новой задачи | Установлен. Перед переходом должен быть сброшен. | Установлен. Перед вызовом должен быть сброшен. | Не меняется. Перед возвратом должен быть установлен. |
| Флаг B старой задачи. | Сбрасывается. | Не меняется, он уже установлен | Сбрасывается. |
| Флаг NT новой задачи | Не меняется. | Устанавливается. | Восстанавливается из TSS новой задачи. |
| Флаг NT старой задачи | Не меняется. | Не меняется. | Сбрасывается. |
| Поле Link новой задачи. | Не меняется. | Загружается селектором TSS старой задачи. | Не меняется. |
| Поле Link старой задачи. | Не меняется. | Не меняется. | Не меняется. |
| Флаг TS в регистре CR0. | Устанавливается. | Устанавливается. | Устанавливается. |

Глава 8. Задачи и страничная организация памяти. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_tss/chap_7.htm;j=false;s=1280*1024;d=24;rand=0.6842944589670545

        Мультизадачность и страничная организация памяти могут работать независимо друг от друга, но максимальный эффект от применения защищённого режима достигается именно при совмещении этих технологий; они не конфликтуют между собой и поэтому вы можете при построении операционной системы сначала отладить взаимодействие ОС со страницами, а потом переходить к работе с задачами либо наоборот.   
        И всё же, есть одно ограничение при использовании задач и страниц. Если граница страниц находится внутри TSS, т.е. когда TSS пересекает границу двух или более страниц, то нужно быть внимательным, потому что процессор при работе с TSS использует преобразование линейного адреса в физический один раз, когда вычисляет адрес начала TSS. Если вторая страница будет не присутствовать или отображена не на соседнюю физическую, то процессор, перейдя границу страниц, не будет вычислять физический адрес новой, а продолжит обращаться по смежному физическому адресу. В результате будут загружены не значения полей TSS, а другая информация и это приведёт либо к нарушению контекста задачи, либо, скорее всего, к исключению. Поэтому, рекомендация - размещайте TSS целиком внутри одной страницы, либо на смежных страницах.   
        Во избежание этой проблемы и для упрощения работы с сегментами состояния задач, возможны два варианта:

|  |  |
| --- | --- |
| 1. | Для содержимого каждого TSS, кроме его дескриптора, определять отдельный сегмент данных, в котором он будет располагаться и отображать такой сегмент на смежные страницы памяти. |
| 2. | TSS всех задач, работающих в системе, размещать в одной области памяти. В наших примерах для этого определяется сегмент данных TSS\_area, но он определяется размером в 4Кб и целиком занимает одну страницу, т.к. в примерах определяется не много задач и их TSS имеют минимальные размеры. Если в вашей ОС может использоваться много задач и они будут иметь достаточно большие размеры, например, хранить в себе карты ввода/вывода и прочую информацию, то определяя такой сегмент, отображайте его явно на последовательную цепочку страниц, чтобы физические адреса, на которые будет отображён этот сегмент, были последовательны и непрерывны для всего сегмента. |

        Применяя эти два варианта или их комбинацию, можно создать устойчивую к сбоям структуру задач. Например, можно определить в одном сегменте все системные TSS - они, как правило, не имеют карты ввода/вывода и могут быть минимального размера, в другом сегменте - все TSS драйверов, для них подразумеваются карты ввода/вывода, и, наконец, в третьем сегменте - TSS всех прикладных задач - им "не положены" карты ввода/вывода и они все имеют подобную структуру.   
        Говоря о сегменте для TSS, мы подразумеваем **алиасный** сегмент данных, отображённый на те же адреса памяти, по которым располагается TSS, определённый дескриптором TSS. Т.к. обращаться к памяти через дескриптор TSS нельзя (это системный объект), то необходимо создавать дополнительный сегмент данных - тут мы поступаем аналогично определению LDT для задач, когда таблицы LDT всех задач расположены одна за другой в одной области памяти, которую описывает один сегмент данных. На рис. 8-1 поясняется такое расположение LDT и TSS:

  
**Рисунок 8-1. Пример размещения всех таблиц LDT и всех сегментов TSS.**

        До сих пор говорилось о том, что контекст задачи целиком размещается в её TSS. Это - простой вариант, но не самый эффективный. Контексты модулей FPU, MMX и XMM как правило сохраняются и загружаются целиком одной соответствующей командой и наиболее быстро это будет происходить, когда адрес начала области, предназначенной для хранения контекста какого-либо из этих модулей, будет выравнен на границу линии кэша. Линия кэша - это элемент данных минимального размера, процессор при записи в память либо чтении из неё оперирует блоками памяти в 32 байта (в Pentium 4 - 64 байта). Поэтому, выравнивая адрес начала области памяти для контекста модулей FPU, MMX и XMM на границу линии кэша (т.е. на 32 или 64 байта), вы предотвращаете лишние циклы шины, что повышает производительность всей системы.   
        Сегменты состояния задачи имеют минимальный размер в 104 байта, т.е. они не кратны 32 или 64 байтам, поэтому размещая контексты устройств в TSS вы должны выбирать между потерями на лишние циклы шины и экономией памяти под TSS. В таком случае лучше будет хранить контексты FPU, MMX и XMM в отдельных от TSS областях памяти - это повысит производительность, сократит размер TSS и упростит управление задачами.

Глава 9. Реализация мультизадачности. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_tss/chap_8.htm;j=false;s=1280*1024;d=24;rand=0.9631413516604344

        Для начала вашему вниманию предлагается пример, в котором определяется три задачи - "Main", "2" и "3". Задача Main производит переключения на задачи 2 и 3 командой FAR CALL, а задачи 2 и 3 возвращают управление в Main командой IRET. Этот пример демонстрирует инициализацию задач и простейший способ управления ими. Исключения и прерывания здесь не реализованы в виде отдельных задач, поэтому при возникновении исключения или прерывания, обработчик будет выполняться в контексте текущей задачи.   
        Для того, чтобы увидеть мультизадачность в действии, три задачи выполняют простейшие действия:

|  |  |
| --- | --- |
| 1. | Main\_TSS: Выводит строку "First task switches other tasks:" |
| 2. | TSS\_1: Выводит строку "Second Task: Increment:" и число, которое увеличивается на 1 при каждом переключении на задачу. |
| 3. | TSS\_2: Выводит строку "Third Task: Decrement:" и число, которое уменьшается на 1 при каждом переключении на эту задачу. |

        Для работы мультизадачности необходимо следующее:

|  |  |
| --- | --- |
| 1. | Подготовить все необходимые дескрипторы. |
| 2. | Установить контексты всех задач в их сегментах TSS. |
| 3. | Запустить мультизадачность и обеспечить переключение задач. |

        В примере мы определим следующие 4 дескриптора:

|  |  |
| --- | --- |
|  | TSS\_area - сегмент данных, в котором определены TSS всех задач. |
|  | Три дескриптора TSS: Main\_TSS, TSS\_2 и TSS\_3. |

        Вот так эти дескрипторы определяются (перед переходом в защищённый режим):

; Устанавливаем сегмент TSS\_area размером 4Кб для описаний TSS.

mov bx,offset GDT + TSS\_area

mov eax,cr3

add eax,3 \* 4096 ; EAX - базовый адрес сегмента, ему

; будет выделена не используемая до сих

; пор страница памяти, 3-я после страницы

; для каталога страниц.

mov ebp,eax ; Временно сохраняем этот адрес.

mov edx,4096 ; Предел сегмента

mov cx,92h ; CL - байт прав доступа,

; CH - биты GDXU и старшие 4 бита предела.

call set\_descriptor ; TSS\_area

; Устанавливаем дескрипторы трёх TSS:

mov bx,offset GDT + Main\_TSS

mov eax,ebp

mov edx,67h

mov cx,10001001b

call set\_descriptor ; Main\_TSS

add eax,104

call set\_descriptor ; TSS\_2

add eax,104

call set\_descriptor ; TSS\_3

        После входа в защищённый режим, мы определяем контексты всех используемых задач. В принципе, это можно было сделать ещё и в режиме реальных адресов, но, располагая код инициализации задач в защищённом режиме, мы повышаем защищённость системы.   
        Все TSS, используемые в данном примере, имеют одинаковый минимальный размер и подобную структуру, поэтому для установки TSS используется процедура "set\_TSS":

set\_TSS proc near

; Создаёт 32-разрядный TSS минимального размера (104 байта - предел 67h).

; DI = указатель на TSS внутри сегмента TSS\_area

; Параметры новой задачи:

; EAX = CR3

; EBX = EIP

; ECX = EFLAGS

; EDX = ESP

push eax

push bx

push di

push es

push eax ; Сохраняем регистры для очистки TSS.

push ecx

push di

mov ax,TSS\_area ; ES:DI указывают на начало TSS.

mov es,ax

xor eax,eax ; Очищаем 26 двойных слов (т.е. 104 байта)

mov cx,26 ; в TSS. Это делается для потому, что

cld ; большинство полей TSS будут содержать нули.

rep stosd

pop di

pop ecx

pop eax

mov es:[ di + 28 ],eax ; Записали CR3

mov es:[ di + 32 ],ebx ; EIP

mov es:[ di + 36 ],ecx ; EFLAGS

mov es:[ di + 56 ],edx ; ESP

mov al,8

mov es:[ di + 76 ],al ; CS ( = 8 = Code\_selector )

add al,8

mov es:[ di + 80 ],al ; SS ( = 16 = Stack\_selector )

add al,8

mov es:[ di + 84 ],al ; DS ( = 24 = Data\_selector )

pop es

pop di

pop bx

pop eax

ret

endp

        Вот так выглядит инциализация задач:

; Установка мультизадачности

xor di,di ; Это будет первая задача (Main\_TSS).

call set\_TSS ; Для задачи Main можно вообще не

; определять TSS. Почему - см. далее.

add di,104 ; Это будет вторая задача (TSS\_2).

lea ebx,second\_task\_entry\_point ; EIP задачи.

mov edx,esp ; ESP 2-й задачи.

sub edx,256

call set\_TSS

add di,104 ; Это будет третья задача (TSS\_3).

lea ebx,third\_task\_entry\_point ; EIP задачи.

mov edx,esp ; ESP 3-й задачи.

sub edx,512

call set\_TSS

        Сразу после такого определения контекстов задач можно запускать мультизадачность. Посмотрите, как это делается в этом примере:

; Запускаем первую задачу:

mov ax,Main\_TSS

ltr ax ; Загружаем в регистр TR селектор

; дескриптора TSS задачи Main. Теперь текущая

; задача - это Main\_TSS.

; Чистим Busy flag Main\_TSS - он установился после загрузки селектора в TR

mov bx,offset GDT + Main\_TSS

and byte ptr [ bx + 5 ],11111101b

; Переход на Main\_TSS

db 0eah ; Этот код эквивалентен команде FAR JMP Main\_TSS:00

dw 00

dw Main\_TSS

; Вот здесь процессор сохранил контекст текущей задачи, т.е. заполнил

; поля TSS текущими значениями.

; Main\_TSS:

; А теперь процессор загрузил те же самые значения из TSS в регистры.

; Вот почему можно не определять TSS для первой задачи - процессор всё

; равно перезапишет в него текущие значения, однако тогда нужно быть

; уверенным, что поля для SSi:ESPi будут содержать нули.

; Установка Busy flag Main\_TSS - она нужна, потому что был переход с

; Main\_TSS на Main\_TSS командой JMP - эта команда сбрасывает флаг занятости

; старой задачи, т.е. Main\_TSS.

mov bx,offset GDT + Main\_TSS

or byte ptr [ bx + 5 ],10b

; Вывод строки первой задачи.

mov text\_color,1fh

lea bx,fst\_task\_msg

mov dx,0600h

call put\_zs

mtss\_1:

db 9ah ; far call TSS\_1:00

dw 00

dw TSS\_2

db 9ah ; far call TSS\_1:00

dw 00

dw TSS\_3

jmp mtss\_1

second\_task\_entry\_point:

mov text\_color,1ah

lea bx,snd\_task\_msg

mov dx,0700h

call put\_zs

xor eax,eax

step\_1:

mov text\_color,1ah

call put\_dd\_num

inc eax

sub dl,8

iret

jmp step\_1

third\_task\_entry\_point:

mov text\_color,1eh

lea bx,trd\_task\_msg

mov dx,0800h

call put\_zs

xor eax,eax

ttep\_1:

mov text\_color,1eh

call put\_dd\_num

dec eax

sub dl,8

iret

jmp ttep\_1

        Рабочий пример доступен здесь: файлы examp\_9.asm, examp\_9.com, pmode\_9.lib и init.lib в архиве examp\_9.zip (19'586 байт).

        В этом примере был слегка изменён обработчик аппаратного прерывания клавиатуры, так, что программа реагирует на нажатие клавиши Esc и таким образом можно прервать её выполнение. Здесь не акцентируется внимание на том, как именно был дополнен обработчик прерывания, подробно это будет описано в разделе, посвящённом программированию устройств.

        Из-за обилия подключаемых процедур, строки макросов, их подключающих, теперь будут вынесены в отдельный файл "init.lib", а код инициализации перед переходом в защищённый режим и после перехода вынесены в две отдельные процедуры. Файл examp.asm теперь содержит только те процедуры, которые обсуждались в данном разделе - это упрощает понимание исходного текста.

Глава 10. Использование LDT. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_tss/chap_9.htm;j=false;s=1280*1024;d=24;rand=0.5159473590196036

        В этой главе и в примере, приведенном в конце, показано, как используются локальные дескрипторные таблицы в задачах.   
        В качестве базового примера используются предыдущий - те же три задачи, переключаются через FAR CALL из основной задачи и программа прекращает работу по нажатию Esc. Обработчики исключений и прерываний также не реализованы отдельными задачами - пример предназначен для демонстрации использования LDT.   
        Для реализации мультизадачности нужно выполнить те же три действия:

|  |  |
| --- | --- |
| 1. | Подготовить все необходимые дескрипторы. |
| 2. | Установить контексты всех задач в их сегментах TSS. |
| 3. | Запустить мультизадачность и обеспечить переключение задач. |

        В каждом из них будут некоторые изменения, по сравнению с предыдущим примером.

        В примере используются следующие дескрипторы:

|  |  |
| --- | --- |
|  | TSS\_area - описывает алиасный сегмент данных, отображённый на сегменты состояния всех задач. |
|  | LDT\_area - описывает алиасный сегмент данных, отображённый на локальные дескрипторные таблицы всех задач. |
|  | Task\_Stack - сегмент данных, используемый как стек для всех задач. Каждой задаче будет выделена область в этом сегменте. |
|  | Main\_TSS, TSS\_2 и TSS\_3 - дескрипторы TSS всех трёх задач. |
|  | LDT\_2 и LDT\_3 - дескрипторы LDT задач 2 и 3 (задача Main не использует LDT). |

        Вот так выглядит код инициализации дескрипторов. Этот код выполняется до перехода в защищённый режим.

; Устанавливаем сегмент TSS\_area размером 4Кб для описаний TSS.

mov bx,offset GDT + TSS\_area

mov eax,cr3

add eax,3 \* 4096

mov ebp,eax ; Временно сохраняем этот адрес.

mov edx,4096

mov cx,92h

call set\_descriptor ; TSS\_area

; Устанавливаем сегмент LDT\_area размером 4Кб для описаний LDT.

mov bx,offset GDT + LDT\_area

mov eax,cr3

add eax,4 \* 4096

mov edx,4095

mov cx,92h

call set\_descriptor ; LDT\_area

; Устанавливаем сегмент Task\_stack размером 4Кб для всех стеков задач.

mov eax,cr3

add eax,5 \* 4096

mov edx,4095

mov cx,92h

call set\_descriptor ; Task\_stack

; Устанавливаем дескрипторы трёх TSS:

mov bx,offset GDT + Main\_TSS

mov eax,ebp

mov edx,67h

mov cx,10001001b

call set\_descriptor ; Main\_TSS

add eax,104

call set\_descriptor ; TSS\_2

add eax,104

call set\_descriptor ; TSS\_3

; Устанавливаем дескрипторы двух LDT:

mov bx,offset GDT + LDT\_2

mov eax,cr3

add eax,4 \* 4096 + 64

mov edx,65

mov cx,10000010b

call set\_descriptor ; LDT\_2

add eax,64

call set\_descriptor ; LDT\_3

        Процедура установки TSS немного усложнилась. Теперь в ней добавилось следующее:

|  |  |
| --- | --- |
|  | Значение поля IOPL берётся из образа EFLAGS для задачи, переданного в регистре ECX. Это поле используется для построения селекторов и определяет уровень привилегий задачи. |
|  | Если задача использует LDT (т.е. регистр BP при вызове процедуры содержит 0), то задача будет использовать локальные селекторы с установленным битом TI и полем RPL, равным значению поля IOPL. |
|  | Конструируется LDT задачи размером 64 байта (т.е. для 8 дескрипторов) и в ней определяются три дескриптора - кода, стека и данных. Содержимое дескрипторов кода и стека копируется из соответствующих дескрипторов в GDT, а для стека используется копия дескриптора Task\_Stack. На самом деле для задач нужно создавать новые, уникальные дескрипторы, но в данном примере акцент делается на использовании дескрипторов LDT, а не их создании. |

        Перед вызовом этой процедуры регистр EBP должен содержать для создаваемой задачи:

|  |  |
| --- | --- |
|  | в старшей половине - указатель в LDT\_area на начало описания LDT, |
|  | в младшей половине - селектор дескриптора LDT. |

set\_TSS proc near

; Создаёт 32-разрядный TSS минимального размера (104 байта - предел 67h).

; DI = указатель на TSS внетри сегмента TSS\_area

; Параметры новой задачи:

; EAX = CR3

; EBX = EIP

; ECX = EFLAGS

; EDX = ESP

; EBP = LDT\_description & LDT\_sel

push eax

push bx

push di

push es

push eax

push ecx

push di

mov ax,TSS\_area

mov es,ax

xor eax,eax

mov cx,26

cld

rep stosd ; Очистка TSS перед заполнением.

pop di

pop ecx

pop eax

mov es:[ di + 28 ],eax ; Записали CR3

mov es:[ di + 32 ],ebx ; EIP

mov es:[ di + 36 ],ecx ; EFLAGS

mov es:[ di + 56 ],edx ; ESP

mov es:[ di + 96 ],bp ; LDT

mov eax,ecx ; EAX = ECX = EFLAGS

shr eax,12

and al,11b ; AL = IOPL

mov dl,al ; Временно сохраняем значение IOPL в DL.

cmp bp,0

je stss\_1

or al,100b ; TI = 1 - селектор выбирает дескрипторы из LDT.

stss\_1:

add al,8

mov es:[ di + 76 ],al ; CS

add al,8

mov es:[ di + 80 ],al ; SS

add al,8

mov es:[ di + 84 ],al ; DS

cmp bp,0

je stss\_2

; Установка LDT задачи размером 64 байта (т.е. на 8 дескрипторов).

mov ax,LDT\_area

mov es,ax

shr ebp,16

sub bp,offset LDT\_area

mov di,bp ; ES:DI = указатель на LDT задачи

xor eax,eax

mov cx,16

push di

rep stosd ; Очищаем 64 байта LDT.

pop di

add di,8 ; Пропускаем 0-й дескриптор.

; Записываем дескриптор сегмента кода (1-й дескриптор в этой LDT). Т.к.

; код задачи находится в том же сегменте, что и основной код программы, то

; мы просто создадим в LDT копию 1-го дескриптора (доступного по селектору

; Code\_selector).

lea bx,GDT

add bx,Code\_selector

mov eax,[ bx ]

stosd

mov eax,[ bx + 4 ]

stosd

; Память, предназначенная для стека описывается для всех задач одним

; сегментом - Task\_Stack. Его и будет использовать задача в регистре SS:

lea bx,GDT

add bx,Task\_Stack

mov eax,[ bx ]

stosd

mov eax,[ bx + 4 ]

stosd

; Создаём копию дескриптора данных:

lea bx,GDT

add bx,Data\_Selector

mov eax,[ bx ]

stosd

mov eax,[ bx + 4 ]

stosd

stss\_2:

pop es

pop di

pop bx

pop eax

ret

endp

        Установка мультизадачности также усложняется, потому что теперь для каждой задачи нужно указывать больше параметров:

; Установка мультизадачности

xor di,di ; Это будет первая задача.

mov eax,cr3

lea ebx,first\_task\_entry\_point ; EIP задачи.

mov ecx,10b ; EFLAGS задачи.

mov edx,256 - 4 ; ESP 1-й задачи.

mov bp,0 ; LDT у 1-й задачи нет.

call set\_TSS

add di,104 ; Это будет вторая задача.

lea ebx,second\_task\_entry\_point ; EIP задачи.

mov edx,512 - 4 ; ESP 2-й задачи.

mov bp,offset LDT\_area + 64

shl ebp,16

mov bp,LDT\_2 ; LDT 2-й задачи.

call set\_TSS

add di,104 ; Это будет третья задача.

lea ebx,third\_task\_entry\_point ; EIP задачи.

mov edx,768 - 4 ; ESP 3-й задачи.

mov bp,offset LDT\_area + 128

shl ebp,16

mov bp,LDT\_2 ; LDT 3-й задачи.

call set\_TSS

        Вот и всё. Полностью пример - файлы examp\_10.asm, examp\_10.com, pmode\_10.lib и init.lib, - доступен в архиве examp\_10.zip (19'843 байт).

Глава 11. Шлюз задачи. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_tss/chap_10.htm;j=false;s=1280*1024;d=24;rand=0.05025021578640454

        Одним из основных условий, обеспечивающих защиту в защищённом режиме, является изоляция программ на разных уровнях привилегий. Программа, работающая на уровне привилегий 3 (это менее привилегированный уровень), может обращаться к сегментам данных, кода и системным объектам, находящихся только на том же уровне привилегий. Тоже самое касается программ, работающих на уровнях 2 и 1 и только программам на нулевом уровне привилегий (самым привилегированным или, иначе говоря, системным программам) можно обращаться к любым сегментам и объектам, за одним исключением - для любого уровня привилегий передача управления может производитьсятолько на том же уровне привилегий. Уровень привилегий задаётся в двухбитовом поле DPL дескриптора сегмента или системного объекта; теоретическое назначение и практическое использование уровней привилегий подробно обсуждается в разделе "Защита".   
        В общем виде, операционная система использует несколько уровней привилегий. Когда происходит прерывание или исключение, процессор обращается к дескрипторной таблице прерываний IDT, выбирает из неё соответствующий дескриптор и передаёт управление обработчику прерываний или исключения. Таблица прерываний в системе одна, обработчики находятся на 0-м уровне привилегий. Что будет, если прерывание или исключение произойдёт в программе, работающей на не нулевом уровне привилегий? Ведь процессор, по идее, не позволяет передачу управления между уровнями привилегий, а переключение через IDT на обработчик прерывания как раз и является передачей управления из одной программы (процедуры) в другую, причём, они могут находится на разных уровнях привилегий.   
        Итак, процессор не позволяет прямую межуровневую передачу управления и этим изолирует процедуры на разных уровнях привилегий. Однако, процессор позволяет передачу управления через посредника - специальный объект, так называемый шлюз. Функция шлюза сводится к тому, что запрос на передачу управления от менее привилегированной процедуры может "опуститься" на более привилегированный уровень, а после того, как запрошенная процедура отработает, управление возвратится начальной процедуре, "поднявшись" на свой уровень привилегий через этот же шлюз.   
        Шлюзы специально предусмотрены для межуровневой передачи управления; подразумевается, что созданием шлюзов должна заниматься только операционная система. Именно из шлюзов состоит дескрипторная таблица прерываний, в ней разрешено применять три типа таких объектов - шлюзы прерывания, ловушки и задачи. Первые два подробно рассматриваются в разделе "Прерывания в защищённом режиме", в этой главе мы рассмотрим шлюз задачи.   
        Шлюз задачи позволяет установить в качестве обработчика вектора прерывания отдельную задачу. Формат шлюза задачи следующий:

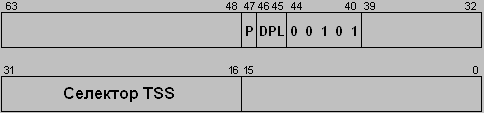
dw 0 ; Зарезервировано

dw TSS\_sel ; Селектор дескриптора TSS задачи обработчика.

db 0 ; Зарезервировано

db access\_rights ; Права доступа

dw 0 ; Зарезервировано

  
**Рисунок 11-1. Схема шлюза задачи.**

        Как видите, вся информация хранится в двух полях:

|  |  |
| --- | --- |
|  | **Селектор TSS** - указывает на дескриптор TSS задачи, *на которую* произойдёт переключение, если передача управления будет происходить через этот шлюз. |
|  | **Права доступа:**   |  |  | | --- | --- | |  | **флаг P** (присутствие шлюза) определяет наличие шлюза в памяти. Как, правило этот флаг всегда установлен. | |  | **поле DPL** определяет численно наибольший уровень привилегий, который должна иметь процедура, обращающаяся к обработчику прерывания через команды INT n, INT 3 и INTO. Если обращение к шлюзу задачи вызвано исключением или аппаратным прерыванием, то это поле DPL игнорируется. | |

        Реализуя мультизадачную систему, следует позаботиться о том, чтобы все используемые вектора прерываний обрабатывались отдельными задачами. Вообще говоря, можно не определять эти задачи и тогда обработка прерываний и исключений будет происходить в контексте текущей задачи (например, так это было в двух предыдущих примерах). И всё же, реализация всех обработчиков или наиболее важных из них отдельными задачами значительно повышает устойчивость системы к сбоям, происходящим как по вине ошибок в самой системе, так и умышленных попыток нарушить целостность ОС.   
        Каждая задача обработчика имеет свой контекст, в котором определены все сегменты, используемые им в работе, в том числе, такие критические, как код и стек. Контекст задачи позволяет полностью изолировать обработчик прерывания от всех остальных программ и это послужит гарантией надёжности его работы.

Глава 12. Использование шлюзов задачи. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_tss/chap_11.htm;j=false;s=1280*1024;d=24;rand=0.7954651356688205

        В этой главе приводится пример реализации мультизадачной системы, в которой переключение задач происходит через прерывания. Для этого обработчик исключения прерывания таймера (IRQ 0) реализован в виде отдельной задачи и переключение задач производится по прерыванию таймера (примерно 18 раз в секунду).   
        В дескрипторе TSS прерванной задачи флаг занятости B остаётся установленным и одной из функций задачи "Main", переключающей две другие задачи "2" и "3", является явный сброс это флага в дескрипторе TSS только что прерванной задачи.   
        Задачи 2 и 3 не "подразумевают" о том, что их прерывают, они не передают управление другим задачам и такой вариант работы ОС является наиболее общим видом мультизадачной системы.   
        Пример, приведенный в конце главы, использует следующую схему работы:

|  |  |
| --- | --- |
|  | Определяютcя 4 задачи: Main, 2, 3 и Timer; задача Main производит инициализацию мультизадачности, задачи 2 и 3 являются рабочими задачами, а Timer обеспечивает поочерёдное переключение между задачами 2 и 3. |
|  | Задача Main разрешает прерывание от таймера (которое было запрещено при инициализации примера) и зацикливается. Больше никаких действий эта задача выполнять не будет - как только сработает прерывание таймера, управление в задачу Main уже не вернётся. |
|  | В примере реализована очередь из двух задач Task\_list, представляющая собой два далёких адреса перехода на задачу 2 и 3, расположенных последовательно один за другим. Переменная Task\_number хранит номер текущей задачи: Task\_number = 0/1 - текущей задачей является задача "2" / "3". |
|  | Обработчик прерывания таймера изменён следующим образом:   |  |  | | --- | --- | |  | прерывания запрещены ещё при построении контекста задачи. Это гарантирует, что во время работы задачи Timer, она не будет прервана другими аппаратными прерываниями. С другой стороны, простота самой задачи гарантирует то, что во время выполнения кода задачи не возникнет исключения; | |  | после посылки команды конца прерывания, задача Timer, с помощью переменной Task\_number, определяет текущую задачу и сбрасывает в её дескрипторе TSS флаг занятости B; | |  | в конце цикла работы задачи Timer, управление передаётся следующей задаче в очереди задач. Для передачи управления используется команда FAR JMP. | |

        Таким образом, схема работы задач в такой системе получается следующей:

Timer\_TSS → TSS\_1 → Timer\_TSS → TSS\_2 → Timer\_TSS → TSS\_1 → Timer\_TSS → TSS\_2 → ...

        или так:

[ [ Timer\_TSS -> TSS\_i ] = n раз ] = ? раз,

где "n" означает число задач в системе, а "?" - то, что этот цикл повторяется любое число раз.

        В такой схеме только один "минус" - то, что при каждом переключении задач предварительно запускается задача Timer, что понижает производительность системы, однако, наличие такой интерфейсной задачи позволяет операционной системе не только управлять переключением задач, но и выполнять следующие действия:

|  |  |
| --- | --- |
|  | Обрабатывать таймер - менять соответствующим образом переменные времени, даты и некоторые системные счётчики, зависящие от времени. |
|  | Использовать простую структуру рабочих задач и простой способ управления ими, что повышает надёжность и защищённость системы (например, в ОС можно реализовать переключение задач только через очередь и только такой интерфейсной задачей (Timer) и тогда можно запретить задачам самостоятельно вызывать другие задачи). |
|  | Менять дополнительный контекст задачи, например, загружать контекст модуля FPU, MMX и/или XMM до того, как задача явно запросит систему об этом (через исключение), что сэкономит время и, соответственно, повысит производительность системы. |
|  | Можно перед запуском следующей в очереди задачи проводить некоторые проверки, позволяющие повысить производительность системы. Так, например, задачи, находящиеся в состоянии "сна" (stand by) можно пропускать; также можно пропускать задачи, ожидающие событий от устройств ввода, но ещё не получивших нужные данные (например, пропустить задачу, ожидающую ввода с клавиатуры, если с клавиатуры ничего не было введено). |

        Предлагаемый вашему вниманию пример похож на предыдущий, но имеет следующие дополнения:

|  |  |
| --- | --- |
|  | В GDT добавлен дескриптор "Timer\_TSS", описывающий контекст задачи обработчика прерывания таймера. |
|  | Установка дескриптора TSS для Timer\_TSS производится вместе с остальными дескрипторами TSS:  ; Устанавливаем дескрипторы трёх основных TSS:  mov bx,offset GDT + Main\_TSS  mov eax,ebp  mov edx,67h  mov cx,10001001b  call set\_descriptor ; Main\_TSS  add eax,104  call set\_descriptor ; TSS\_2  add eax,104  call set\_descriptor ; TSS\_3  ; Устанавливаем дескриптор Timer\_TSS:  mov bx,offset GDT + Timer\_TSS  add eax,104  call set\_descriptor ; Timer\_TSS |
|  | Установка мультизадачности также немного отличается от предыдущего примера:  ; Установка мультизадачности  xor di,di ; Это будет первая задача.  mov eax,cr3  mov ecx,202h ; EFLAGS задачи с установленным флагом  ; прерываний - они будут разрешены.  mov edx,100 ; ESP 1-й задачи.  mov bp,0 ; LDT у 1-й задачи нет.  call set\_TSS  add di,104 ; Это будет вторая задача.  lea ebx,second\_task\_entry\_point ; EIP задачи.  mov edx,200 ; ESP 2-й задачи.  mov bp,offset LDT\_area + 64  shl ebp,16  mov bp,LDT\_2 ; LDT 2-й задачи.  call set\_TSS  add di,104 ; Это будет третья задача.  lea ebx,third\_task\_entry\_point ; EIP задачи.  mov edx,300 ; ESP 3-й задачи.  mov bp,offset LDT\_area + 128  shl ebp,16  mov bp,LDT\_2 ; LDT 3-й задачи.  call set\_TSS  add di,104 ; Это задача таймера.  lea ebx,Timer\_task\_entry\_point ; EIP задачи.  mov ecx,10b ; EFLAGS задачи таймера - прерывания будут  ; запрещены.  mov edx,400 ; ESP задачи таймера.  mov ebp,0 ; LDT у задачи таймера нет.  call set\_TSS  ; Запускаем первую задачу:  mov ax,Main\_TSS  ltr ax  ; Чистим Busy flag Main\_TSS  mov bx,offset GDT + Main\_TSS  and byte ptr [ bx + 5 ],11111101b  ; Переход на Main\_TSS  db 0eah  dw 00  dw Main\_TSS |
|  | Вот так задача Main инициализирует в примере мультизадачность с прерыванием по таймеру:  ; Main\_TSS:  mov ax,Data\_Selector  mov ds,ax  ; Установка Busy flag Main\_TSS  mov bx,offset GDT + Main\_TSS  or byte ptr [ bx + 5 ],10b  mov text\_color,1fh  lea bx,fst\_task\_msg  mov dx,0600h  call put\_zs  mov Task\_number,0  ; Разрешаем прерывания от таймера:  in al,21h  and al,11111110b  out 21h,al  ; Зацикливаем эту задачу. Теперь по прерыванию таймера управление  ; будет передано в задачу Timer\_TSS и больше сюда не возвратится.  mtss\_1:  jmp mtss\_1  second\_task\_entry\_point:  mov text\_color,1ah  lea bx,snd\_task\_msg  mov dx,0700h  call put\_zs  xor eax,eax  step\_1:  mov text\_color,1ah  call put\_dd\_num  inc eax  sub dl,8  jmp step\_1  third\_task\_entry\_point:  mov text\_color,1eh  lea bx,trd\_task\_msg  mov dx,0800h  call put\_zs  xor eax,eax  ttep\_1:  mov text\_color,1eh  call put\_dd\_num  dec eax  sub dl,8  jmp ttep\_1          Обратите внимание, что рабочие задачи (это "2" и "3") обе работают в бесконечном цикле и не передают управление другим задачам. Переключение задач действительно происходит по таймеру. |
|  | Вот так выглядит обработчик прерывания от таймера; этот код является кодом задачи Timer:  IRQ\_0\_handler macro  Timer\_task\_entry\_point:  push ax  mov al,20h  out 20h,al  pop ax  lea bx,Task\_List  mov al,Task\_number  mov dl,al ; Временно сохраняем это значение.  cmp al,1  je timer\_1  inc al  add bx,4  jmp timer\_2  timer\_1:  dec al  timer\_2:  mov Task\_number,al  ; Очистка флага занятости у прерванной задачи:  ; 1. Вычисляем дескриптор TSS прерванной задачи.  lea si,GDT  mov dh,0  shl dx,3 ; DX = DL \* 8  add dx,TSS\_2 ; Теперь DX равен селектору:  ; TSS\_2, если Task\_number был равен 0  ; TSS\_3, если Task\_number был равен 1  add si,dx ; DS:SI указывают в GDT на содержимое  ; дескриптора TSS.  ; 2. Сброс флага занятости.  and byte ptr [ si + 5 ],11111101b  ; Переключение на новую задачу:  jmp dword ptr [ bx ]  jmp Timer\_task\_entry\_point  endm |
|  | И, наконец, прекращение работы программы произойдёт по нажатию клавиши Esc, для чего ещё в прошлых примерах соответствующем образом изменён обработчик клавиатуры. |

        Сам пример доступен здесь - файлы examp\_11.asm, examp\_11.com, pmode\_11.lib и init.lib, - в архиве examp\_11.zip (20'501 байт).