Руслан Аблязов

Программирование на ассемблере на платформе x86-64

Москва, 2011

УДК 004.438Assembler
ББК 32.973.26-018.1

Аблязов Р. З.
А13 Программирование на ассемблере на платформе x86-64. – М.: ДМК Пресс, 2011. – 304 с.: ил.

ISBN 978-5-94074-676-8

В данной книге речь идёт о работе процессора в двух его основных режимах: защищённом режиме и 64-битном, который также называют long mode («длинный режим»). Также помимо изложения принципов и механизмов работы процессора в защищённом и 64-битном режимах, речь пойдёт о программировании на ассемблере в операционных системах семейства Windows, как в 32-битных, так и 64-битных версиях. Рассматривается не только разработка обычных приложений для операционных систем Windows, но и разработка драйверов на ассемблере. При написании книги уделялось большое внимание именно практической составляющей, т.е. изложение материала идёт только по делу и только то, что необходимо знать любому системному и низкоуровневому программисту. Последний раздел книги посвящён принципам работы многопроцессорных систем, а также работе с расширенным программируемым контроллером прерываний (APIC).

На диске, прилагаемом к книге, находятся полные исходные коды примеров к книге, а также дополнительные программы и материалы.

Издание предназначено для системных и низкоуровневых программистов, а также для студентов и преподавателей технических специальностей высших и средне-специальных учебных заведений .

УДК 004.438Assembler
ББК 32.973.26-018.1

ISBN 978-5-94074-676-8

Содержание

Используемый компилятор

При компиляции всех примеров, приведённых в этой книге, необходимо использовать FASM. Почему я выбрал именно этот компилятор? По очень многим причинам.

1. Максимальный набор поддерживаемых команд. FASM поддерживает весь (или почти весь) набор команд x86-64.
2. Наличие Linux- и Windows-версий, а также поддержка широкого списка выходных файлов. Можно компилировать программы для Windows (формат PE) и для Linux (формат ELF).
3. Гибкий синтаксис и отсутствие совершенно бесполезных, но обязательных директив (например, .386, .486 и т. п.).
4. Полный контроль над размещением данных в исполняемом файле.
5. Мощнейший макросный движок, с помощью которого можно создать практически любой макрос и изменить текст программы и сам язык до неузнаваемости.
6. При компиляции через командную строку нет необходимости указывать множество опций компиляции. Для того чтобы получить исполняемый файл нужного типа, достаточно задать все необходимые параметры в самом исходнике.
7. Windows-версия FASM поставляется вместе с IDE, благодаря которой можно скомпилировать программу нажатием одной кнопки (или пункта меню).

Компилятор FASM можно найти на компакт-диске, прилагающемся к книге (папка fasm). Последнюю версию этого компилятора также можно бесплатно скачать на сайте flatassembler.net.

1 Основы

1.1. Основные понятия

Поскольку книга посвящена программированию на платформе x86-64 и все примеры в книге приведены на языке ассемблер, то было бы правильным первый раздел книги посвятить изложению основ программирования на ассемблере. Даже неподготовленный читатель после прочтения первого раздела книги сможет усвоить материал следующих разделов.

Ниже будет рассказано о принципе работы процессора, а также основных понятиях, необходимых при программировании на ассемблере и не только на ассемблере. Под *процессором* подразумеваются процессоры от Intel и его клоны (разумеется, главными клонами являются процессоры AMD). Итак, начнём.

1.1.1. Что такое процессор?

Что такое процессор? *Процессор* – это важнейшая часть любой компьютерной системы, именно на нём и происходит вся вычислительная деятельность (или почти вся).

Любая компьютерная система состоит из трёх частей: центральный процессор, системная память и внешние устройства. Все три компонента взаимодействуют друг с другом посредством шин; наиболее важная из них – системная шина: именно с её помощью процессор взаимодействует с системной памятью и с важнейшими контроллерами. Здесь нам понадобится ввести четвёртое понятие – *контроллер*. Контроллер может выполнять разные действия: быть посредником между процессором и внешними устройствами (контроллер ввода-вывода, контроллер прерываний, USB-контроллер и т. д.) либо выполнять некоторые специфичные операции (например, контроллер прямого доступа к памяти).

Основная задача памяти и контроллеров в целом понятна; какова же задача процессора? Основная задача процессора – это выполнение инструкций, которые находятся в памяти. Всё время после включения компьютера и до самого выключения процессор выполняет инструкции – за исключением некоторых случаев, когда он переходит в «подвешенное» состояние для экономии энергии. В зависимости от режима, в котором находится процессор, инструкции он выполняет поразному.

Что такое инструкция? *Инструкция*, или *машинная команда*, – это просто некоторый код, фактически обычное число, которое находится в памяти и обозначает некоторое действие. Инструкции бывают разные и разного размера – они могут

занимать от одного до нескольких байт. По сути, они дают указание процессору, что ему надо делать. Для того чтобы процессор «знал», откуда из памяти брать команды для выполнения, он использует специальный указатель инструкции. Выполнив очередную инструкцию, процессор обновляет этот указатель, так чтобы он указывал на следующую команду в памяти, и т. д. Если какая-либо инструкция не может быть выполнена, процессор генерирует исключение; если исключение не может быть обработано, то он перезагружается.

На рис. 1.1 приведена наиболее упрощённая схема устройства процессорной системы.

Линии, которыми соединяются блоки, – это шины: шина адреса, шина данных и шина управления. Все они объединяются в одну системную шину. Для расширения возможностей системы на системную шину «цепляются» контроллеры. Каждый контроллер исполняет команды, которые были посланы ему через системную шину.

Самая главная составляющая любой системы – это *память* (системная память). Память бывает двух видов: физическая и линейная.

Процессор тоже имеет свою память двух видов: кэш-память и регистровую. Дефакто *кэш-память* – это такая же системная память, просто находящаяся внутри кристалла процессора. Кэш-память является копией некоторой области основной памяти, к которой он часто обращается.

Регистровую память представляют собой регистры. *Регистр* – это такое устройство, которое хранит в себе некоторую информацию, т. е. некоторое значение. Разрядность значения определяет разрядность регистра. Одни регистры могут хранить только определённую информацию, другие – любую. Те регистры, которые могут хранить любую информацию, называются *регистрами общего назначения*. Остальные регистры напрямую управляют работой процессора; точнее сказать, сами они

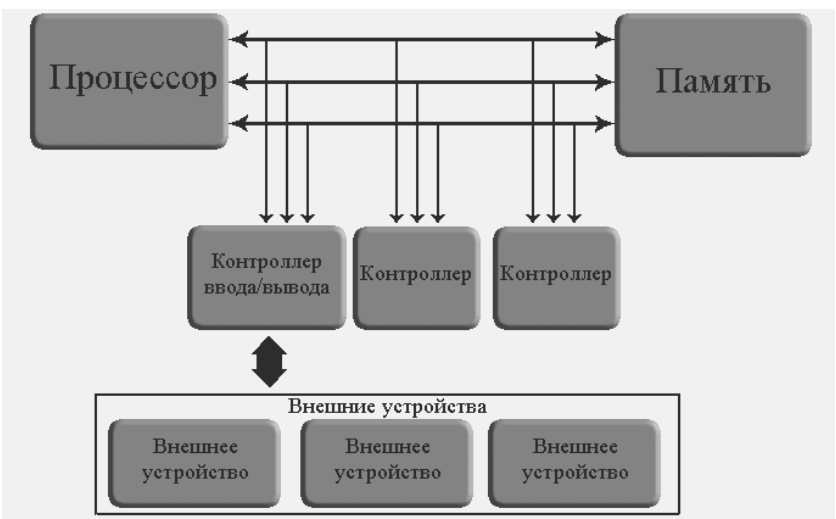


Рис. 1.1. Упрощённая схема устройства компьютера

не управляют, а процессор работает по-разному в зависимости от значений, которые принимают эти регистры. Регистры бывают разной разрядности, но большинство регистров 8-, 16-, 32- и 64-разрядные.

Для того чтобы получить значение, которое находится по определённому адресу памяти, процессор выставляет на шину адреса физический адрес, после чего на шине данных появляется значение. Конечно, появляется не мгновенно, но очень быстро – это зависит от скорости работы системной шины, памяти и других параметров. Если на шину адреса выведется значение памяти по адресу примерно большее 512 Мб, а на машине реально установлено 256 Мб, то в зависимости от материнской платы и памяти на шину данных может выйти просто «ерунда» или нулевое значение. Точно так же устанавливается и значение ячейки памяти. Вы спросите: «Как же материнская плата "узнает", что собирается сделать процессор?». На шину управления выставляется определённое значение, которое указывает материнской плате (или модулю памяти), что процессор хочет писать в память или читать из неё.

Всё вышесказанное само по себе вряд ли нам сможет пригодиться, но эта информация важна для понимания материала, который будет приводиться ниже. Обращения к определённым адресам памяти (которых обычно реально нет в системной памяти) могут перехватываться некоторыми котроллерами и перенаправляться к устройствам. Перехватив обращение к памяти, контроллер перенаправляет данные, которые должны быть помещены в память (или считаны из памяти) некоторому устройству. Таким образом, программа может взаимодействовать с внешними устройствами посредством записи/чтения из определённых адресов памяти. Это основной метод взаимодействия программ и устройств: например, так происходит взаимодействие с PCI-устройствами.

Как уже было сказано выше, адрес, который выставляется на шину адреса, – это и есть физический адрес. С виртуальной памятью всё будет сложнее. Виртуальная память может и не существовать реально, т. е. она «вроде бы есть, но её как бы нет». Если приложение операционной системы обращается к несуществующему адресу, процессор транслирует его в существующий адрес, а если памяти не хватает, то другая занятая память выгружается на внешнюю память (HDD, Flash и т. д.). Более подробно о виртуальной памяти будет говориться в следующих разделах, посвящённых программированию в защищённом режиме.

Как же происходит выполнение программ на процессоре? Все данные, которыми может оперировать процессор, находятся в памяти и только в памяти; если данные находятся не в памяти, а на внешних устройствах, то необходимо сначала загрузить эти данные в память и только потом с ними работать. Программа, дефакто, – это тоже данные, просто они представляют собой коды инструкций. Код инструкции называют *опкодом*. Опкод – это несколько байтов данных (от 1 до 10 и более), закодированных специальным образом, чтобы процессор мог понять, что от него «хотят». Иначе говоря, опкоды инструкций – это приказы процессору, которые тот беспрекословно выполняет, например: переслать данные из одного регистра в другой, из регистра в память, выполнить вычитание или сложение и т. д.

Процессор использует специальный регистр в качестве указателя инструкции EIP (IP, RIP). Этот регистр всегда содержит адрес следующей инструкции. После выполнения очередной инструкции процессор читает инструкцию из адреса, на который указывает регистр EIP, определяет её размер и обновляет адрес в регистре EIP (прибавляет к значению в регистре EIP размер текущей инструкции). После выборки инструкции из памяти и изменения регистра EIP процессор приступает к её выполнению. Инструкция может выполняться от одного до нескольких десятков процессорных тактов. В процессе выполнения инструкции может измениться адрес в регистре EIP. При изменении адреса в регистре EIP следующей будет выполнена именно та инструкция, адрес которой содержит регистр EIP – иными словами, произойдёт переход, или прыжок. После выполнения инструкции процессор читает следующую инструкцию, которая находится по адресу, содержащемуся в регистре EIP, и процесс повторяется. Вся сущность работы процессора заключается в выполнении команд. Однажды включившись, он всегда должен выполнять команды до тех пор, пока он не выключится или не перейдёт в специальное «подвешенное» состояние (ждущий режим).

Ещё одно важное понятие – это *стек*. Стек – это специальная область памяти, которая используется для хранения промежуточных данных. Представьте, что одна подпрограмма вызывает другую; вызываемая подпрограмма завершила свое выполнение, и теперь ей надо передать управление подпрограмме, которая её вызвала. Адрес команды, к которой надо вернуться после выполнения вызванной подпрограммы, находится на верхушке стека.

На рис. 1.2 приведён пример цепочки вызова подпрограмм и содержимого стека.

Стек – как магазин автомата: добавили адрес, добавили второй, добавили третий, и затем адреса достают в порядке, обратном добавлению: третий, второй, первый. Следовательно, подпрограмма может вызвать ещё одну подпрограмму, а та, в свою очередь, – ещё одну, и так сколько угодно – всё ограничивается только размером стека. На верхушку стека указывает определённый регистр; более подробно о стеке мы будем говорить в разделе 2.

На рис. 1.2 адреса возврата выделены рамкой. *Адрес возврата* – это адрес следующей инструкции после инструкции вызова (CALL). При выполнении команды возврата из подпрограммы (RET) адрес возврата извлекается из верхушки стека и происходит передача управления по этому адресу.

1.1.2. Небольшая предыстория

Исторически сложилось так, что все процессоры, которыми мы пользуемся, называются процессорами x86. Под процессорами x86 подразумевают следующие модели процессоров: 8086, 80186, 80286, 80386, 80486, 80586 (Pentium I), 80686 (Pentium II) и т. д. Процессоры 8086-80286 – 16-разрядные, все остальные – 32-разрядные. Процессоры Intel с поддержкой технологии EM64T и процессоры AMD с поддержкой технологии AMD64 являются 64-разрядными процессорами. Фактически технологии AMD64 и EM64T – это одно и то же; разработчиком этой технологии является компания AMD, а Intel всего лишь «лицензировала» (если это можно так назвать) технологию AMD64 у компании AMD.

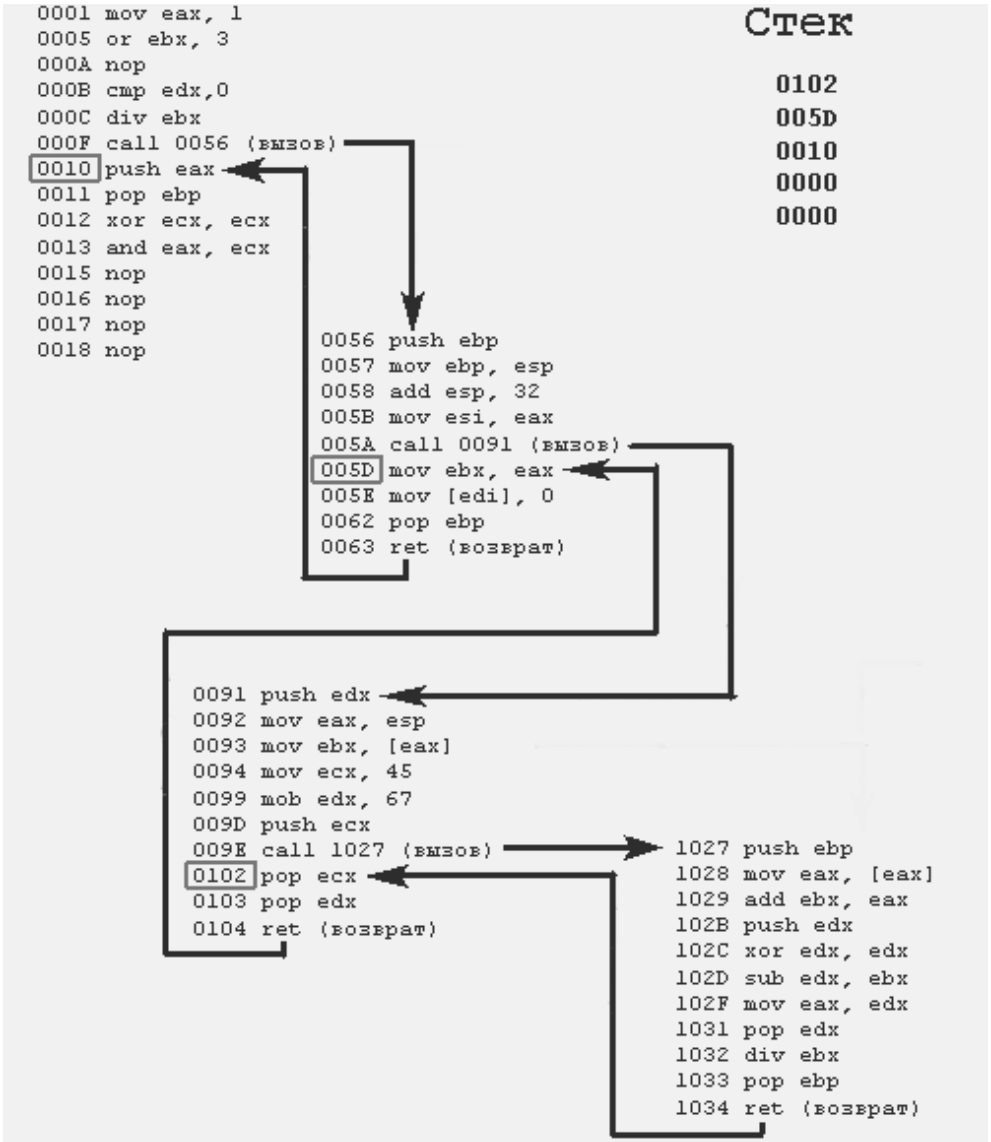


Рис. 1.2. Пример цепочки вызова подпрограмм и содержимое стека

Компания Intel тоже разработала свою 64-разрядную технологию под названием IA-64, но она настолько инновационная, что для её описания не хватит всей этой книги. По этой технологии компания Intel разработала свой процессор Itanium. Архитектура IA-64 (и процессоры Itanium) главным образом создавалась для использования на высокопроизводительных серверных системах; в этой книге она рассматриваться не будет.

Платформы, базирующиеся на процессорах с поддержкой AMD64 или EM64T, мы в дальнейшем будем называть x86-64 платформами. Часто платформа x86-64 упоминается под названием x64, но это более узкое понятие, которое подразумевает только 64-разрядный режим процессора. Платформа x86-64 полностью совместима с x86 и является её логическим развитием и продолжением. Платформы, базирующиеся на процессорах Itanium, будем называть IA-64. Не следует путать x86-64 и IA-64: это совершенно разные технологии.

1.1.3. Процессоры x86-64

Поскольку сейчас в основном выпускают только процессоры с поддержкой технологии AMD64 (EM64T), в этой книге речь пойдёт главным образом о работе процессоров x86-64. По сути, процессоры x86-64 – это почти все процессоры, которые можно встретить на рынке и при повседневном использовании.

Процессоры x86-64 в основном могут работать в трёх основных режимах: реальный режим, защищённый режим и 64-разрядный режим, или long mode (далее – long mode):

- *реальный режим* – это режим, в который переходит процессор после включения или перезагрузки. Это стандартный 16-разрядный режим, в котором доступно только 1 Мб физической памяти и возможности процессора почти не используются, а если и используются, то в очень малой степени. Иногда этот режим называют режимом реальных адресов, потому что в нем нельзя активировать механизм трансляции виртуальных адресов в физические. Это значит, что все адреса, к которым обращаются программы, являются физическими, т. е. без какого-либо преобразования будут выставлены на шину адреса. В этом режиме «родной» для процессора размер равен 2 байтам, или слову (WORD);
- *защищённый режим* (protected mode, или legacy mode по документации AMD) – это 32-разрядный режим; разумеется для процессоров x86 этот режим главный. В защищённом режиме 32-разрядная операционная система может получить максимальную отдачу от процессора – разумеется, если ей это потребуется. В этом режиме можно получить доступ к 4-гигабайтному физическому адресному пространству, если память, конечно, установлена на материнской плате, а при включении специального механизма трансляции адресов можно получить доступ к 64 Гб физической памяти. В защищённый режим можно перейти только из реального режима. Защищённый режим называется так потому, что позволяет защитить данные операционной системы от приложений. В этом режиме «родной» для процессора размер данных – это 4 байта, или двойное слово (DWORD). Все операнды, которые выступают в этом режиме как адреса, должны быть 32-битными;
- long mode («длинный режим», или IA-32e по документации Intel) – это собственно сам 64-разрядный режим. По своему принципу работы он почти полностью сходен с защищённым режимом, за исключением нескольких аспектов. В этом режиме можно получить доступ к 2^{52} байтам физической памяти и к 2^{48} байтам виртуальной памяти. В 64-разрядный режим можно

перейти только из защищённого режима. В этом режиме «родной» для процессора размер данных – это двойное слово (DWORD), но можно оперировать данными размером в 8 байт. Размер адреса всегда 8-байтовый.

Помимо приведённых выше режимов есть ещё один режим. Это режим системного управления (System Management Mode), в который процессор переходит при получении специального прерывания SMI. Режим системного управления предназначен для выполнения некоторых действий с возможностью их полной изоляции от прикладного программного обеспечения и даже операционной системы. Переход в этот режим возможен только аппаратно. Режим системного управления может использоваться для реализации системы управления энергосбережением компьютера или функций безопасности и контроля доступа.

Помимо вышеперечисленных режимов работы процессор поддерживает следующие два подрежима:

- *режим виртуального процессора 8086* – это подрежим защищённого режима для поддержки старых 16-разрядных приложений. Его можно включить для отдельной задачи в многозадачной операционной системе защищённого режима;
- *режим совместимости для long mode*. В режиме совместимости приложениям доступны 4 Гб памяти и полная поддержка 32-разрядного и 16-разрядного кода; «родной» для процессора размер данных – это двойное слово. Режим совместимости, можно сказать, представляет собой в long mode то же самое, что и режим виртуального 8086 процессора в защищённом режиме. Режим совместимости можно включить для отдельной задачи в многозадачной 64-битной операционной системе. В режиме совместимости размер адреса 32-битный, а размер операнда не может быть 8-байтовым.

Итак, мы узнали, в каких режимах может работать процессор, но в официальной документации производителей процессоров (Intel и AMD) немного другая терминология. Если следовать официальной документации, есть два режима работы процессора: 32-битный и 64-битный. 32-битный режим в документации от AMD называется legacy mode, а в документации от Intel носит название IA-32. Он включает в себя режим реальных адресов и защищённый режим. 64-битный режим в документации от AMD называется long mode, в документации от Intel – IA-32e; он включает в себя два подрежима: сам 64-разрядный режим и режим совместимости.

На рис. 1.3 изображены диаграмма режимов работы процессора и возможности перехода из одного режима в другой.

Из схемы видно, что в 64-битный режим можно перейти, только лишь предварительно включив защищённый режим. После перезагрузки процессора, в каком бы он режиме ни находился, он всё равно начнёт работать в режиме реальных адресов. Непосредственный переход в режим реальных адресов возможен только из защищённого режима, а находясь в других режимах это придется делать только посредством перезагрузки процессора.

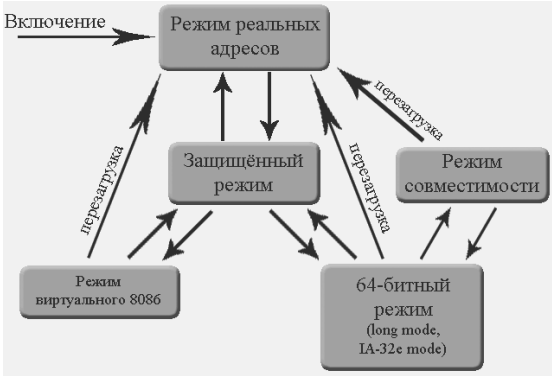


Рис. 1.3. Диаграмма режимов процессоров x86-64

В этой книге режим виртуального процессора 8086 и режим реальных адресов описаны не будут – литературы, где описывается программирование на ассемблере в режиме реальных адресов, существует предостаточно. Мы же рассмотрим только защищённый режим и 64-битный режим процессора.

1.1.4. Регистры процессоров x86-64

В защищённом режиме, в режиме реальных адресов и режиме совместимости доступны следующие регистры:

- регистры общего назначения: 32-разрядные EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP; 16-разрядные AX, BX, CX, DX, SI, DI, SP, BP (они являются младшими частями 32-разрядных регистров); 8-битные регистры AH, BH, CH, DH и AL, BL, CL, DL (старшие и младшие части 16-битных регистров соответственно);
- 32-разрядный EIP (IP в реальном режиме) – указатель инструкции;
- 16-разрядные сегментные регистры: CS, DS, SS, ES, FS, GS;
- 32-разрядный регистр флагов – EFLAGS;
- 80-битные регистры математического сопроцессора ST0-ST7 и др.;
- 64-битные MMX-регистры – MM0 – MM7;
- 128-разрядные XMM-регистры – XMM0 – XMM7 и 32-битный MXCSR;
- 32-разрядные регистры управления CR0 – CR4; регистры-указатели системных таблиц GDTR, LDTR, IDTR и регистр задачи TR;
- 32-разрядные регистры отладки – DR0 – DR3, DR6, DR7;
- MSR-регистры.

В режиме реальных адресов доступны не все вышеуказанные регистры, но регистры управления доступны в любом случае. В режиме реальных адресов нельзя использовать некоторые регистры размером более 16 бит.

При переключении процессора в 64-разрядный режим программе доступны следующие регистры:

- регистры общего назначения: 64-разрядные RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP и R8, R9, ..., R15; 32-разрядные EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, R8D – R15D (являются младшими частями 64-разрядных регистров); 16-разрядные AX, BX, CX, DX, SI, DI, SP, BP, R8W – R15W (являются младшими частями 32-разрядных регистров); 8-битные регистры AH, BH, CH, DH и AL, BL, CL, DL, SIL, DIL, SPL, BPL, R8L – R15L (старшие и младшие части 16-битных регистров соответственно);
- 64-разрядный RIP – указатель инструкции;
- 16-разрядные сегментные регистры: CS, DS, SS, ES, FS, GS;
- 64-разрядный регистр флагов – RFLAGS;
- 80-битные регистры математического сопроцессора ST0 – ST7;
- 64-битные MMX-регистры (MM0 – MM7);
- 128-разрядные XMM-регистры – XMM0 – XMM15 и 32-битный MXCSR;
- 64-разрядные регистры управления CR0 – CR4 и CR8; регистры-указатели системных таблиц GDTR, LDTR, IDTR и регистр задачи TR;
- 64-разрядные регистры отладки – DR0 – DR3, DR6, DR7;
- MSR-регистры.

Теперь немного пояснений. Регистры сегментов напрямую участвуют в формировании адресов, каждый сегментный регистр указывает на свой сегмент памяти, а именно: CS – сегмент кода, DS – сегмент данных, SS – сегмент стека; остальные три регистра дополнительные и могут не использоваться программой. Свободная работа с ними не всегда возможна; например в защищённом и 64-разрядном режимах загружать в них можно лишь определённые значения. В защищённом и 64-разрядном режимах доступность регистров зависит уровня привилегий, на котором выполняется программа.

Регистр общего назначения ESP (RSP) всегда указывает на верхушку стека, но при этом нам ничто не мешает использовать его в других целях, хотя тогда будет потеряна возможность нормальной работы со стеком. Вообще все регистры общего назначения можно свободно использовать в своих целях, но следует помнить, что некоторые регистры используются некоторыми командами: например, EBP (RBP) обычно указывает на начало фрейма в стеке, где хранятся локальные данные подпрограмм.

Указатель инструкции EIP (RIP) напрямую использовать нельзя – данный регистр используется самим процессором.

Регистры STn, MMn, XMMn используются математическим сопроцессором при работе с числами с плавающей точкой.

Регистры CRn, DRn, регистры-указатели системных таблиц, MSR-регистры являются системными и управляются ключевыми механизмами работы процессора.

Не следует сейчас заострять внимание на регистрах, их названиях и назначении: более подробно о регистрах будет рассказано ниже.

1.1.5. Память

Начнём с начала – с 70-х годов прошлого века. Тогда компьютер с памятью 64 Кб считался суперкомпьютером – если она была установлена, это было вообще чудо.

Для адресации такой памяти хватало 16 бит. После того как появились модули памяти 128 Кб и даже 256 Кб, регистров размером 16 бит стало не хватать для адресации памяти. Тогда ввели 16-битные сегментные регистры. Вся память делилась на сегменты размером 64 Кб. Сегмент задавался сегментным регистром, а адрес – обычным регистром или непосредственно 16-битным значением адреса. Но даже после этих новаций можно было адресовать только 1 Мб физической памяти из-за особенности формирования физического адреса и особенностей архитектуры процессора.

Всё кардинально изменилось, когда был выпущен первый полноценный 32-разрядный процессор (Intel 80386): он мог работать в двух режимах – в обычном 16-разрядном, как старые процессоры, и в защищённом, в котором можно было адресовать до 4 Гб физической памяти. На процессоре Intel 80386 появились новые 32-разрядные регистры, с помощью которых можно было адресовать эту память; также он поддерживал набор команд, позволяющий работать с данными размером 32 бита. Сегментные регистры остались, но они не слишком сильно влияли на формирование адреса, а предназначались для защиты определённых областей памяти, которые могла задавать операционная система; отсюда и название этого режима. Впрочем, для того чтобы получить доступ к 4 Гб памяти, необязательно надо было переходить в защищённый режим: такая возможность существовала и в режиме реальных адресов, но защищённый режим давал намного больше новых возможностей, чем просто расширение физического адресного пространства до 4 Гб.

После выпуска первых процессоров с поддержкой технологии AMD64 появился и 64-битный режим, который являлся вариантом защищённого режима, рассчитанным только на 64-битную архитектуру. В нём важность сегментных регистров была снижена до минимума, а защита данных операционной системы полностью возложена на механизм трансляции страниц.

В чем суть этого изложения краткой истории развития процессорной техники? Суть в том, чтобы показать, какие бывают адреса. Существует три вида адресов:

1. Физический.
2. Линейный (или виртуальный).
3. Логический.

С первым пунктом всё понятно: физический адрес – это адрес в системной памяти компьютера, именно тот адрес, который выставляется на шину адреса. Самое сложное – это третий пункт. Но начнём по порядку.

Для того чтобы получить доступ к некоторому значению в памяти (в любом режиме), приложение должно указать сегмент и адрес в этом сегменте. Сегмент указывается в сегментном регистре или непосредственно значением (это значение может быть только 16-битным), а адрес – в обычном регистре или непосредственно значением (это значение может быть 16-, 32-, 64-битным в зависимости от режима). Приложение может указать адрес разными способами, а сегмент можно вовсе не указывать. Если он не указан, то значение сегмента берётся из соответствующего (код, данные или стек) сегментного регистра. Хотите вы этого или

нет, адрес всегда будет в таком формате – «сегмент:смещение»; именно этот адрес и называется логическим.

После преобразования логического адреса (способ преобразования тоже зависит от режима процессора) получается абсолютный 20-, 32-, 64-битный адрес в зависимости от режима; этот адрес называется линейным. В режиме реальных адресов он сразу выставляется на шину адреса. В защищённом и 64-разрядном режимах этот адрес можно называть виртуальным, если активирован механизм трансляции страниц (подробнее об этом механизме рассказано в разделе 2.3). Если механизм трансляции страниц не активирован, то линейный адрес становится физическим, т. е. без преобразования выставляется на шину адреса. Если же механизм трансляции страниц включён, то линейный (он же виртуальный) адрес специальным образом преобразуется в физический адрес; способ преобразования задаётся самой операционной системой.

1.1.6. Работа с внешними устройствами

Одна из основных задач процессора – это взаимодействие с внешними устройствами. Если процессор не будет взаимодействовать с внешними устройствами, то какая от него польза? Теперь следует привести схему процессорной системы, немного отличную от той, которая была рассмотрена выше, в разделе 1.1.1 (см. рис. 1.4).

Все стрелки на схеме – это шины; некоторые из них находятся на системной шине, а некоторые – на шинах, связывающих устройства с процессором.

Процессор взаимодействует с внешними устройствами через порты ввода/вывода (не путайте их с портами LPT, PS/2 и т. д.) и через спроецированные на память регистры устройств. Это взаимодействие основано на том, что процессор выводит какие-либо данные в порты либо в определённые ячейки памяти, на которые спроецированы регистры устройств. На вышеприведённом рисунке посредником между процессором и устройствами выступает контроллер ввода-вывода (это общий случай; иногда посредником между процессором и внешним устройством может выступать какой-либо другой контроллер).

Но как процессор «узнаёт» о том, что устройство обработало данные или получило новые данные от пользователя? Если внешнее устройство хочет что-то «сказать» процессору (например, что оно обработало запрос или приняло от пользователя какие-либо данные), то оно через контроллер прерываний передаёт процессору сигнал о прерывании.

Итак, мы ввели новое понятие – *прерывание*. Прерывание – это сигнал процессору о том, чтобы он прервал выполнение текущей программы и передал управление специальной функции, называемой функцией-обработчиком прерывания. Программа через контроллер ввода-вывода получает необходимые данные для обработки этого запроса.

1.1.7. Резюме

В этом разделе мы познакомились с понятием «процессор», изучили основные режимы и принципы его работы. В разделе 1.2 речь пойдёт об основах ассемблера.

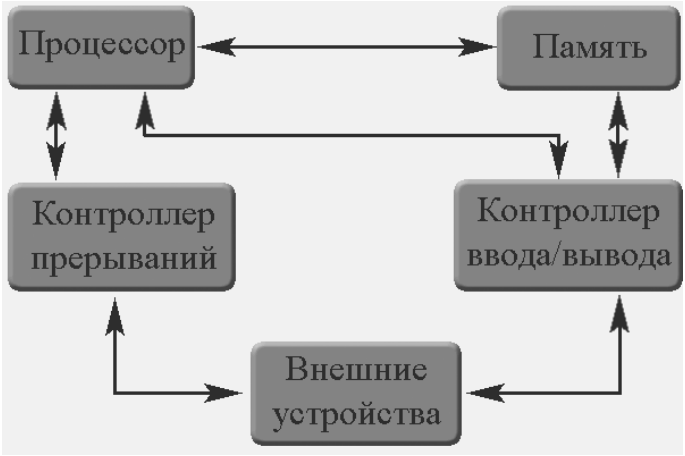


Рис. 1.4. Работа процессора с внешними устройствами

1.2. Основы ассемблера

Когда вы пишете программу на ассемблере, вы фактически пишете команды процессору. Команды процессору – это просто коды (или коды операций, или опкоды). Из-за этого ассемблер считается самым низкоуровневым языком программирования: все в ассемблере непосредственно преобразовывается в шестнадцатеричные коды. Другими словами, у вас нет компилятора, который преобразовывает язык высокого уровня в язык низкого уровня – ассемблер (программа) переводит команды, понимаемые человеком, непосредственно в шестнадцатеричные коды.

В этом разделе мы обсудим несколько базовых команд, которые имеют отношение к вычислениям, поразрядным операциям и т. д. Другие команды (команды перехода, сравнения и пр.) будут рассмотрены позже.

1.2.1. Немного о языке ассемблера

В данной книге будет использоваться ассемблер FASM. Все примеры к разделам книги будут написаны с использованием компилятора FASM. С учётом этого при описании команд будет учтена специфика синтаксиса FASM.

В программах на ассемблере каждая команда должна находиться на одной строке, т. е. одна строка – одна команда. Комментарии в ваших программах оставляются после точки с запятой; все символы строки, которые находятся после точки с запятой, считаются комментарием.

Числа в ассемблере могут представляться в двоичной, десятеричной или шестнадцатеричной системе. Для того чтобы показать, в какой системе использовано число, надо поставить после него букву. Для бинарной (двоичной) системы пишется буква b (пример: 0000010b, 001011010b), для десятеричной системы можно ничего не указывать после числа или добавить в конце букву d (примеры: 4589, 2356d), для шестнадцатеричной системы надо добавить в конце букву h, запись шестнадцатеричного числа начинается с нуля, за которым следуют буквы A–F

(примеры: правильно 0AC45h; неправильно F145Ch, C123h). Также шестнадцатеричные числа можно записывать в форматах языков высокого уровня: с начальным знаком $, как в pascal, или сочетанием 0x, как в языке C. Символы значения могут быть заключены как в кавычки, так и в апострофы.

Идентификаторы в программе не могут содержать символов `+-*/=<>()[]{}:,|&~#``. В этой книге чаще всего будут использоваться шестнадцатеричные числа, причём отсчёт чисел будет всегда начинаться с нуля (например, если речь идёт про пятый бит, значит, это шестой бит в привычном понимании).

1.2.2. Регистр флагов

Если после выполнения команды произошла ошибка, то, следовательно, все дальнейшие команды будут ошибочны, что может привести к нежелательным последствиям. Для того чтобы указать программе о результате выполнения очередной команды, процессор должен сохранить результат в некотором регистре. Этот регистр называется *регистром флагов* EFLAGS (FLAGS, RFLAGS). Регистр флагов является ключевым в работе процессора: он управляет основополагающими аспектами его работы.

После выполнения очередной команды процессор сохраняет результат выполнения команды в этом регистре. Каждый бит этого регистра обозначает определённое состояние или результат. Некоторые биты управляют работой процессора, а некоторые обозначают только лишь результат выполнения предыдущей команды. Например, если после выполнения какой-либо операции (сложения, вычитания, логической операции и пр.) результатом был ноль, то биту, отвечающему за флаг нулевого результата (ZF), будет присвоена единица, в любом другом случае – ноль.

Назначения битов регистра флагов приведены в табл. 1.1.

Названия некоторых флагов говорят сами за себя, некоторые же вообще непонятны. Объяснять здесь назначения флагов не имеет смысла – в дальнейшем мы будем делать это по мере необходимости.

Таблица 1.1. Регистр флагов

Номер бита	Название	Описание
0	CF	Флаг переноса
1	-	Зарезервирован (всегда установлен)
2	PF	Флаг чётности
3	-	Зарезервирован (всегда сброшен)
4	AF	Дополнительный флаг переноса
5	-	Зарезервирован (всегда сброшен)
6	ZF	Флаг нулевого результата
7	SF	Флаг знака результата
8	TF	Флаг трассировки
9	IF	Флаг разрешения прерывания

Таблица 1.1. Регистр флагов

Номер бита	Название	Описание
10	DF	Флаг направления
11	OF	Флаг переполнения
12	IOPL	Текущий уровень привилегий ввода/вывода
13		
14	NT	Флаг вложенности задачи
15	-	Зарезервирован (всегда сброшен)
16	RF	Флаг возобновления
17	VM	Флаг виртуального процессора 8086
18	AC	Флаг проверки выравнивания адреса
19	VIF	Флаг виртуального прерывания
20	VIP	Флаг отложенного виртуального прерывания
21	ID	Флаг разрешения идентификации процессора
22-31(63)	-	Зарезервированы (всегда сброшены)

1.2.3. Команда MOV

Самая первая команда – это команда пересылки данных MOV. Эта команда пересылает данные, а именно байт, слово, двойное слово, четверное слово из источника в назначение. Данная команда является эталоном для всех команд, принимающих два операнда: все остальные команды, работающие с двумя операндами, имеют такой же синтаксис. Команда MOV – самая распространённая, и в большинстве программ до 40% команд являются командами MOV.

Синтаксис команды MOV: `mov` ***<назначение>, <источник>.***

Фактически команда присваивает регистру или памяти некоторое значение, которое можно взять из памяти, регистра или непосредственного значения (просто цифра). Если назначением является регистр, то источником может быть память, регистр или непосредственно значение; если назначение – память, то источником может быть только регистр или непосредственное значение (перемещение из памяти в память недопустимо). Размерность источника и назначения должны быть равными. Примеры присваивания значения регистру приведены в листинге 1.1.

Листинг 1.1. Работа с регистрами при помощи команды MOV

```
mov edx, ebx
mov rax, r10
mov eax, r12d
mov ah, dh
mov cx, r14w
mov ecx, 789056h
mov ax, 12h
mov al, 890h   ; неправильно
mov dx, rbx    ; неправильно
```

Для работы с памятью надо указать адрес и заключить его в квадратные скобки. Адрес должен быть совместим с текущим режимом работы процессора: например, в режиме реальных адресов необходимо указывать 16-битный адрес, а в 64-битном режиме – 64-разрядный адрес. Примеры использования команды MOV для работы с ячейками памяти приведены в листинге 1.2.

Листинг 1.2. Работа с памятью при помощи команды MOV

```
mov eax, dword [0000509Ah]
mov rbx, qword [00A056F1h]
mov dword [0F81562Dh], r9d
mov qword [0F8156F6h], 0DF899564FFh
```

Для того чтобы указать, какой размер данных необходимо пересылать, надо указать перед адресом оператор размера. Операторы размера приведены в табл. 1.2.

Таблица 1.2. Операторы размера

Оператор размера	Размер в байтах
byte	1
word	2
dword	4
fword	6
pword	6
qword	8
tbyte	10
tword	10
dqword	16

Наиболее часто используемые операторы размера – byte, word, dword, qword. Разумеется, в защищённом режиме вместе с командой MOC не получится использовать 8-байтовый операнд, и в режиме реальных адресов не получится использовать 4-байтовый операнд (4-байтовые регистры использовать можно). Некоторые другие команды могут работать с любыми операндами вне зависимости от текущего режима процессора: например, команды математического сопроцессора могут работать с 10-байтовым операндом (tbyte, tword) как в защищённом, так и в 64-битном режиме. Вместо квадратных скобок можно указать директиву PTR перед адресом.

Листинг 1.3. Использование директивы PTR

```
mov eax, dword ptr 0000509Ah
mov rbx, qword ptr 00A056F1h
mov qword ptr 0F81562Dh, r9d
mov qword ptr 0F8156F6h, 09564FFh
```

Перемещение из памяти в память недопустимо. Иногда размер можно не указывать: например, когда мы переносим значение из памяти в регистр, размер регистра уже показывает размерность операнда. Также можно использовать регистр для адресации памяти.

Листинг 1.4. Использование регистра для адресации памяти

```
mov eax, [eax]
mov rbx, ptr ebx
mov [rax], r9d
mov ptr edx, 09564FFh
```

Свободно работать можно только с регистрами общего назначения, а работа с остальными регистрами возможна только вкупе с регистрами общего назначения (отсюда и их название). Т. е. перемещать, присваивать и получать значения можно только через регистры общего назначения. Все остальные регистры (не считая регистров математического сопроцессора) будем называть *привилегированными регистрами,* потому в защищённом режиме доступ к ним можно получить только из привилегированного участка кода. Фактически с привилегированными регистрами может работать только эта команда. Есть также набор специфических для каждого типа машины регистров – регистров MSR. К ним нельзя обратиться с помощью команды MOV, для этого предназначены команды RDMSR (чтение), WRMSR (запись).

Фактически команды перемещения из регистра в регистр и из регистра в память и т. д. – это разные команды, т. е. опкоды у них разные, но они выполняют идентичные действия и поэтому объединены под одним названием. Большинство команд ассемблера не обозначают только один опкод, а объединяют несколько идентичных.

1.2.4. Формат хранения данных в памяти

Приведу пример использования команды MOV. Предположим, у нас есть область памяти:

адрес	275	276	277	278	279	27A	27B	27C	27D	27E	27F	280	281	282
данные	0A	50	32	44	57	25	7A	5E	72	EF	7D	FF	AD	C7

Теперь, допустим, есть команда

```
mov eax, dword [0000027Ah]
```

После выполнения этой команды регистр EAX будет содержать значение 725E7A25h. Возможно, вы заметили, что это инверсия того, что находится в памяти: 25 7A 5E 72. Это происходит потому, что значения сохраняются в памяти, используя формат little endian. Суть в том, что самый младший байт сохраняется в наиболее значимом (т. е. с меньшим адресом) байте: указывается обратный порядок байтов.

Числа в процессоре могут быть только целые: знаковые и беззнаковые. Вещественные числа всё равно представляются в двоичном виде как 32-, 64-, 80-битные значения. Оперировать вещественными значениями как таковыми может только

математический сопроцессор. Работа с вещественными числами требует отдельного рассмотрения и в данной редакции книги описываться не будет.

Как уже было сказано, целые числа могут быть знаковыми и беззнаковыми. Беззнаковые числа представляются обычным преобразованием числа в двоичное; следовательно, диапазон значений беззнаковых чисел: $0...2^n - 1$, где n – это разрядность числа. Старший бит знаковых чисел обозначает знак числа: если старший бит – единица, то число отрицательное. Все остальные биты – это модуль числа. Диапазон знакового числа: $-2^{n-1}...2^{n-1} - 1$. Принципы сложения и вычитания целых, знаковых и беззнаковых чисел одинаковые; отличия возникают при умножении и делении чисел. Более подробно работа с знаковыми и беззнаковыми числами будет описана в разделе 1.4.

1.2.5. Команды SUB и ADD

Команда ADD, как уже понятно из названия, складывает оба операнда и сохраняет полученный результат в операнде назначения. Синтаксис команды ADD: `add <назначение>, <источник>`. Формат операндов идентичен формату операндов команды MOV. Оба операнда складываются, и результат сохраняется в назначении. После операции сложения в зависимости от результата изменяются соответствующие биты в регистре флагов.

Можно догадаться, что команда SUB производит вычитание – а если быть точнее, то из операнда назначения она вычитает операнд источника и сохраняет его в операнде назначения. Синтаксис команды SUB полностью идентичен команде ADD.

Если после сложения (вычитания) двух операндов результат не помещается в операнде назначения (число слишком большое и отрицательное), происходит переполнение и флаг OF устанавливается в 1. Если произошёл перенос или заём из старшего бита (например, из бита 17 или 33), то устанавливается флаг переноса. Обычно эти два флага всегда сопутствуют друг другу, т. е. если устанавливается один, то устанавливается и второй. В случае вычитания может и не установиться флаг переполнения. Но не будем сейчас фокусироваться на этих флагах – более подробно работа с числами и с флагами OF и CF будет рассмотрена в разделе 1.4.

Также есть ещё две команды увеличения и уменьшения операнда – это INC и DEC. Эти команды увеличивают и уменьшают операнд на единицу соответственно. Они принимают только один операнд, в качестве которого может выступать регистр или значение памяти. Размер операнда может быть любым (1, 2, 4, 8 байт).

В качестве параметров всех вышеописанных команд могут выступать как знаковые, так и беззнаковые числа.

1.2.6. Логические операции

Следующие по важности команды – это команды, осуществляющие логические операции. Есть несколько базовых логических операций: операция «или» (команда OR), логическое «и» (команда AND), логическое отрицание (команда NOT), «исключающее или» (команда XOR).

Команда OR – операция битового логического ИЛИ. Синтаксис команды: `or <назначение>, <источник>`. Эта команда производит побитовое логическое сложение между переданными операндами и сохраняет результат в назначении. Операнды могут быть размером 1, 2, 4, 8 байт. При логическом сложении результат является единицей (истиной) при истинности любого из операндов.

Команда AND – операция битового логического И. Синтаксис команды: `and <назначение>, <источник>`. Эта команда производит побитовое логическое умножение переданных операндов и сохраняет результат в назначении. Операнды могут быть размером 1, 2, 4, 8 байт. При логическом умножении результат является единицей (истиной) при истинности обоих операндов.

Команда NOT – операция битового логического отрицания. Синтаксис команды: `not <назначение>`. Эта команда производит побитовое логическое отрицание операнда. При логическом отрицании результат является отрицанием операнда, т. е. если 1, то получается 0, если 0 – то получается 1. Операнд может быть размером 1, 2, 4, 8 байт.

Команда XOR – операция битового исключающего ИЛИ. Синтаксис команды: `xor <назначение>, <источник>`. Команда XOR производит операцию побитового исключающего «или» над переданными операндами и сохраняет результат в назначении. Операнды могут быть размером 1, 2, 4, 8 байт.

Таблица 1.3. Логика команды XOR

Назначение	Источник	Результат
0	0	0
0	1	1
1	0	1
1	1	0

Можно сделать вывод: если операнды одинаковые, то результат 0, если разные – то 1. Таким образом, если указать в качестве обоих операндов один и тот же регистр, то регистр будет обнулён. Этот приём бывает очень полезным при оптимизации кода.

Все четыре вышеперечисленные логические операции в зависимости от результата операции изменяют соответствующие биты в регистре флагов.

1.2.7. Сдвиги

Сдвиг – это побитовый сдвиг операнда вправо или влево. Например, сдвиг числа 11101001 на 3 в результате даст 01001000. Все команды сдвига имеют одинаковый синтаксис: `команда <назначение>, <количество битов>`. Сдвиги бывают разных видов: циклические, арифметические и логические. Начнём с логических.

SHL – логический сдвиг влево. Команда производит сдвиг операнда влево на указанное количество бит. В освободившиеся справа биты заносятся нули. Значение CF совпадает со значением бита, который последним был вытеснен за левый край операнда. Если количество битов не равно 1, то признак переполнения OF не

определен. Если же количество битов равно 1, тогда OF = 0, при том что 2 старших бита исходного значения операнда назначения совпадали; иначе OF = 1.

SHR – логический сдвиг вправо. Команда производит сдвиг операнда вправо на указанное количество битов. В освободившиеся справа биты заносятся нули. Значение CF совпадает со значением бита, который последним был вытеснен за правый край операнда. Если бит знака сохраняет свое значение, то признак переполнения OF = 0, иначе OF = 1. Значение CF совпадает со значением бита, который последним был вытеснен за правый край операнда. Если количество битов не равно 1, то признак переполнения OF не определен. Если же количество равно 1, то флаг OF равен значению старшего бита исходного операнда.

SAL – арифметический сдвиг влево. Команда полностью идентична команде SHL. Это одна и та же команда, только имена разные.

SAR – арифметический сдвиг вправо. Команда идентична команде SHR, за исключением того, что каждый вновь вставленный слева бит равен самому старшему биту изначального операнда. Таким образом, при арифметическом сдвиге вправо операнд не изменит свой знак. Например, сдвиг числа 10101001 на 3 бита вправо в результате даст 11110101.

ROL – циклический сдвиг влево. Команда сдвигает операнд влево на указанное количество битов. Бит, который выходит за левый предел, вставляется справа. Значение CF совпадает со значением бита, который последним был вытеснен за левый край операнда. Если количество битов не равно 1, то признак переполнения OF не определен. Если же количество битов равно 1, то во флаг OF заносится результат выполнения операции исключающего «или», примененной к 2 старшим битам исходного значения операнда.

ROR – циклический сдвиг вправо. Команда сдвигает операнд вправо на указанное количество битов. Бит, который выходит за левый предел, вставляется слева. Значение CF совпадает со значением бита, который последним был вытеснен за правый край операнда. Если количество битов не равно 1, то признак переполнения OF не определен. Если же количество битов равно 1, то в OF заносится результат выполнения операции исключающего «или», примененной к 2 старшим битам результата.

RCL – циклический сдвиг влево через флаг CF. Команда сдвигает операнд влево на указанное количество битов. Бит, который выходит за левый край, заносится во флаг CF, а старое значение CF заносится в освободившийся правый бит. Если количество битов не равно 1, то признак переполнения OF не определен. Если же количество битов равно 1, то в OF заносится результат выполнения операции исключающего «или», применённой к 2 старшим битам результата.

RCR – циклический сдвиг вправо через флаг CF. Команда сдвигает операнд вправо на указанное количество битов. Бит, который выходит за правый край, заносится во флаг CF, а старое значение CF заносится в освободившийся левый бит. Если количество битов не равно 1, то признак переполнения OF не определен. Если же количество битов равно 1, то в OF заносится результат выполнения операции исключающего «или», применённой к 2 старшим битам результата.

У всех команд операнд назначения может быть размером 1, 2, 4, 8 байт.

1.2.8. Работа с флагами процессора

Состояние некоторых наиболее часто используемых битов в регистре флагов можно изменить, используя специально отведённые для этого команды. Ниже перечислены команды, работающие с флагами процессора:

1. CLC – сброс флага переноса (CF=0).
2. CLD – сброс флага направления (DF=0).
3. CLI – сброс флага разрешения прерываний (IF=0).
4. LAHF – сохранение в регистре AH содержимого первого байта регистра флагов (флаги SF, ZF, AF, PF, CF)
5. SAHF – сохранение регистра AH в первый байт регистра флагов (флаги SF, ZF, AF, PF, CF); биты с зарезервированными значениями игнорируются.
6. STC – установка флага переноса в единицу (CF=1).
7. STD – установка флага направления в единицу (DF=1).
8. STI – разрешения прерываний (IF=1).

Все команды, кроме CLI и STI, являются непривилегированными: вы не всегда можете свободно использовать их в своих программах. Нюансы работы с командами CLI и STI будут рассмотрены в следующих разделах книги.

1.2.9. Работа со стеком

На верхушку стека указывает регистр ESP (SP, RSP). Есть две команды, которые работают со стеком: PUSH и POP. Команда PUSH «заталкивает» в стек операнд; если операнд меньше размера элемента стека, то он дополняется нулями. Синтаксис команд PUSH/POP: `push/pop <`***операнд***`>`. Операнд может иметь размер 2, 4, 8 байт в зависимости от режима; также он может быть сегментным регистром. У команды POP операнд не может быть непосредственно значением.

Команда PUSH присваивает памяти, на которую указывает ESP (SP, RSP), значение операнда и уменьшает значение этого регистра на 2, 4 или 8 в зависимости от режима. Команда POP работает аналогично, только наоборот. Чтобы читатель лучше понял принцип работы команд PUSH/POP, приведем их эквиваленты.

Эквивалент команды PUSH:

```
sub esp, 4
mov [esp], <операнд>
```

Эквивалент команды POP:

```
mov <операнд>,[esp]
add esp, 4
```

Приведённые эквиваленты справедливы для операндов размером 32 бита; для операндов размером 16 и 64 бита увеличение и уменьшение значения регистра ESP будет производиться на 2 и 8 байт соответственно. Следует помнить, что в каждом режиме процессора есть свои ограничения на размер операнда, помещаемого в стек.

1.2.10. Резюме

В этом разделе были описаны основные и наиболее важные команды ассемблера. Изучены команда пересылки данных MOV, команда сложения и вычитания, битовых сдвигов и др. В следующем разделе пойдёт речь о данных, метках и переходах.

1.3. Метки, данные, переходы

В этом разделе, как уже понятно из названия, пойдёт речь об использовании переменных в программах на ассемблере, а также о метках и переходах.

Условные и безусловные переходы являются важнейшей частью любой программы; они позволяют писать программы с ветвящимися и циклическими алгоритмами. Без их использования можно написать разве что простейшую программу типа «Hello World!» – при написании любой более сложной программы без использования условных и безусловных переходов обойтись невозможно.

1.3.1. Данные

Данные в программах на ассемблере объявляются (или резервируются) с помощью директив данных. В табл. 1.4 приведены директивы для объявления данных (переменных).

Таблица 1.4. Директивы определения данных и их размеры

Размер (байты)	Определение данных
1	DB
2	DW, DU
4	DD
6	DF, DP
8	DQ
10	DT

За директивой описания данных должно следовать одно или несколько числовых значений, разделенных запятыми. Эти выражения определяют значения для простейших элементов данных, размер которых зависит от того, какая директива используется. Вместо числового значения может стоять символ; впоследствии он будет интерпретирован как числовой код символа (символов). В листинге 1.5 приведены различные способы объявления

Листинг 1.5. Пример использования директив объявления данных

```
db 67h
db 5dh, 0f6h
db "z"      ; то же самое, что и db 7ah
db "w", "k", "y"
dw 8a34h, 0c51h, 8bh
du 9e3ah, 07deh
dw "WE"     ; то же самое, что и db 57h, 45h
dd 01F243D5Eh
dd "WEGa"   ; то же самое, что и db 57h, 45h, 47h, 61h
dq 1122334455667788h
```

За директивами db и du могут следовать строки неопределённого размера; в результате каждый символ будет интерпретирован как его числовое значение. Отличие директивы du от db заключается в том, что каждый символ интерпретируется двумя байтами, старший из которых заполняется нулём, например:

```
db "assembler"
```

то же самое, что и

```
db 61h, 73h, 73h, 65h, 6Dh, 62h, 6Ch, 65h, 72h
du "assembler"
```

то же самое, что и

```
db 0, 61h,0, 73h,0, 73h,0, 65h,0, 6Dh,0, 62h,0, 6Ch,0, 65h,0, 72h
```

Для описания большей последовательности одинаковых данных предназначена директива dup. Она используется после директивы определения данных. Перед ней должно стоять число повторений, а после неё (в скобках) – значение или цепь значений для повторения.

Листинг 1.6. Использование директивы DUP

```
db 7 dup (1Ah)
db 6 dup (45h, 0A3h, 90h)
dd 13 dup (0A713E445h)
dd 9 dup (0A713E445h, 0F8D3E412h)
dw 5 dup (?)
```

Если после директивы определения данных идёт вопросительный знак, то начальное значение этих данных будет не определено. Данные, помеченные вопросительным знаком, называются неинициализированными. Неинициализированные данные не включаются в исполняемый файл и будут доступны лишь после загрузки программы в память (разумеется, если это поддерживается форматом исполняемых файлов, в который будет компилироваться программа). Неинициализированные данные позволяют сократить размер исполняемого файла, т. к. в большинстве случаев не важно какое начальное значение имеет переменная.

1.3.2. Метки

Метка – идентификатор, который используется в программе и возвращает адрес памяти, по которому она находится. Метка – это главная форма взаимодействия с данными в памяти. Если метка используется в программе, то фактически на месте метки будет стоять адрес, на который она указывает.

Существуют разные способы определения меток. Простейший из них – двоеточие после названия метки. За этой директивой на той же строке может следовать инструкция или директива. Она определяет метку, значение которой равно смещению, т. е. адресу точки, в которой она определена. Вернее, смещению следующей директивы или инструкции. Этот метод обычно используется, чтобы пометить места в коде, но ничто не мешает вам пометить этим методом данные.

Данные обычно помечаются другим методом – это написание названия метки перед директивой объявления данных. Метка возвращает адрес данных, перед которыми она объявлена. Компилятор запоминает размер данных, на которые

указывает эта метка, и при использовании с ней операндов, несовместимых по размеру, уведомляет об этом. При использовании этого метода объявления меток отпадает необходимость использования операторов размера (`word`, `dword`, `qword` и т. д.).

Третий метод объявления меток – самый гибкий: это использование директивы `label`. После этой директивы должно следовать имя метки, потом (опционально) размер оператора, далее (тоже опционально) оператор "at" и числовое выражение, определяющее адрес, на который данная метка должна ссылаться. В качестве адреса может использоваться другая метка, ведь метка это то же самое что и адрес, на который она указывает. Примеры объявления меток приведены в листинге 1.7.

Листинг 1.7. Примеры объявления меток

```
metka1:   dd …
data1     db …
      dw …
      …
      mov …
      …
      add …
metka2:
      and …
      xor …
label metka3 dword at metka2
      rcl …
```

Все примеры работы с памятью, которые приводились в предыдущем разделе, не вполне корректны, т. е. в реальных программах выражение типа `mov eax, dword [0000509Ah]` встретить почти невозможно, потому что разработчик программы не в силах рассчитать смещение каждой метки вручную. В большинстве программ для работы с данными в памяти используются метки. В листинге 1.8 приведены примеры их использования.

Листинг 1.8. Примеры использования меток

```
metka1:   dd …
data1     db …
          dw …
      …
      mov al, [data1]
metka2:
      mov ebx,eax
      mov ebx, dword [metka1]
      mov [data1], 4567AADDh
      mov cx, word [metka2]
```

Последняя инструкция – это пример того, что в ассемблере нет разницы между кодом и данными. Назначение последней инструкции: в регистр `cx` помещается опкод инструкции `mov ebx,eax`, в регистре `cx` будет обычное число (код), с помощью которого кодируется инструкция `mov ebx, eax`.

Метка, имя которой начинается с точки, обрабатывается как локальная, и её имя прикрепляется к имени последней глобальной метки (с названием, начинающемся с чего угодно, кроме точки) для создания полного имени этой метки. Так, вы можете использовать короткое имя (начинающееся с точки) где угодно перед следующей глобальной меткой, а в других местах вам придется пользоваться полным именем. Метки, начинающиеся с двух точек, – исключения. Они имеют свойства глобальных, но не создают новый префикс для локальных меток.

Листинг 1.9. Примеры использования локальных и глобальных меток

```
globmetka1:
   .locmetka1 dd …
   .locmetka2:
   mov [.locmetka1], eax
   mov word [.locmetka2], cx
   mov word [.locmetka3], cx          ; ошибка!
   mov word [globmetka2.locmetka2], cx
globmetka2:
   .locmetka3: …
   mov [..globmetka3.locmetka4], cx   ; ошибка!
   mov [globmetka3.locmetka4], cx     ; ошибка!
..globmetka3:
   .locmetka4: dw …
   mov [.locmetka4], cx
```

Метка `$` обозначает текущее смещение или смещение текущей команды; таким образом, чтобы бесконечно зациклить выполнение программы, достаточно написать в программе `jmp $`, т. е. это будет ее безусловный «прыжок на саму себя».

Метки бывают ближние и дальние. Дальность или близость метки определяется только тем, в каком месте она используется. Одна и та же метка, используемая в двух разных командах, в одной команде может быть ближней, а в другой – дальней. Ближняя метка –та, которая определена на далее чем 127 байт после и не далее чем на 128 байт до команды, где она используется. Все метки, которые находятся за этими пределами, являются дальними. Дальность или близость меток имеет значение только с командами передачи управления.

Также при объявлении данных может быть необходимо объявлять разные структуры данных, в то же время находящиеся по одному адресу. Это может пригодиться, когда одни и те же данные в зависимости от контекста могут интерпретироваться по-разному, и удобнее было бы использовать разные названия переменных. В таких случаях применяется директива `virtual`. Далее в листинге 1.10 приведён пример её использования.

Листинг 1.10. Использование директивы VIRTUAL

```
param dd ?
virtual at param
 LowPart dw ?
 HighPart dw ?
end virtual
```

```
virtual at param
 Byte1 db ?
 Byte2 db ?
 Byte3 db ?
 Byte4 db ?
end virtual
```

В примере, описанном в листинге 1.10, обратившись к переменной `LowPart` мы получим младшую часть переменной `param`, а обратившись к переменной `Byte3` получим третий байт переменной `param`, или первый байт переменной `HighPart`. Также с помощью данной директивы можно задавать данные, находящиеся по адресу, который адресует какой-нибудь регистр общего назначения, например:

```
virtual at eax
 Char1 db ?
 Char2 db ?
 Char3 db ?
 Char4 db ?
end virtual
```

Тогда инструкция `mov bl, [Char1]` – то же самое, что и `mov bl, [eax]`, а инструкция `mov bl, [Char3]` – то же самое, что и `mov bl, [eax+2]`. Но более удобным и понятным был бы вариант использования этой директивы, приведённый в листинге 1.11.

Листинг 1.11. Использование VIRTUAL для данных, адресованных регистром

```
    mov eax, string

virtual at eax
 eax.Char1 db ?
 eax.Char2 db ?
 eax.Char3 db ?
 eax.Char4 db ?
end virtual

    mov bl, [eax.Char1]
    mov bl, [eax.Char2]
    mov bl, [eax.Char3]
    mov bl, [eax.Char4]
```

Префикс `eax.` делает код интуитивно понятным и более «читабельным». Использование таких меток не вызывает ошибок при компиляции, т. к. имя регистра EAX является частью имени метки, а символ «точка» рассматривается в данном контексте как обычный символ, подобно любому другому символу.

Более полезной директива `virtual` бывает при использовании регистров для адресации структур данных; об этом будет рассказано в разделе 1.4.

1.3.3. Переходы

Переход – это передача управления другой команде. Фактически переход осуществляется после выполнения каждой команды. В регистре EIP (IP, RIP) находится

адрес команды, которая выполнится следующей. После выполнения команды процессор выполняет команду, находящуюся в памяти, на которую указывает EIP (IP, RIP). Этот регистр доступен только для чтения и изменить его нельзя – он изменяется самим процессором. Но иногда надо выполнить не следующую команду, а команду, которая находится, скажем, через 20 команд. Для этого есть команды переходов. Переходы бывают безусловные и условные.

Есть три основные команды безусловной передачи управления: JMP, CALL и RET. Фактически команда JMP изменяет регистр EIP (IP, RIP) на значение, которое было указано в качестве операнда. Операндом может быть непосредственно значение в памяти, регистр, содержащий адрес, или непосредственно значение адреса. Если привести эквивалент команды `jmp <адрес>`, то он будет такой:

```
mov eip, <адрес>
```

Команда CALL производит переход с сохранением в стеке адреса следующей команды, для того чтобы функция (или процедура), на которую производится переход, могла вернуться назад для дальнейшего выполнения вызвавшего её кода. У этой команды формат такой же, что и у команды JMP. Эквивалент этой команды CALL:

```
push eip
jmp <адрес>
```

Команда RET берёт из верхушки стека адрес возврата и переходит по нему. Она не принимает никаких параметров. Эквивалент команды RET: `pop eip`.

Также существует команда RETN, которая принимает один операнд. Операнд задаёт количество байтов, которое необходимо «вытолкнуть» из стека перед возвращением из процедуры. Эквивалент команды RETN n следующий:

```
pop temp
sub esp, n
mov eip, temp
```

Для дальнего возврата, т. е. возврата, когда произошла межсегментная передача управления, предназначена команда RETF. Инструкция RETF также может принимать параметр, подобно инструкции RETN. Эквивалент инструкции RETF n следующий:

```
pop temp_eip
pop temp_cs
sub esp, n
mov cs, temp_cs
mov eip, temp_eip
```

Если передача (возврат) управления осуществляется на другой уровень привилегий, то происходит переключение стека. Подробнее о переключении стека пойдёт речь в разделе 2.5.

Эквиваленты команд, которые были приведены выше, никогда не сработают, потому что регистр EIP изменить нельзя; примеры были приведены нами только для лучшего понимания предмета.

Команды JMP и CALL могут принимать в качестве операнда как значение памяти, так и непосредственный адрес. Чаще всего в качестве операнда выступает

непосредственно адрес (т. е. метка). В таком случае компилятор анализирует близость адреса и, если он близкий, то генерирует короткий опкод инструкции, в противном случае – длинный опкод инструкции. Т. е. короткий вариант инструкции может генерироваться только тогда, когда операнд не заключён в скобки (или не указано ptr) и не является регистром. Короткий вариант инструкции используется, когда происходит переход не далее чем на 127 байт вперёд и 128 байт назад.

Команды условного перехода передают управление, только если выполнено условие. Таких команд много, и каждая передаёт управление в зависимости от значения некоторого флага в регистре флагов. Например, команда JZ передаёт управление другому адресу, только если выставлен флаг ZF. Команды условного перехода принимают в качестве параметра ближнюю метку, т. е. могут передать управление не далее чем на 127 байт вперёд и 128 байт назад.

Команды условного перехода чаще всего используются вместе с инструкциями сравнения. Чаще всего используемая инструкция сравнения – это команда CMP. Она сравнивает операнды и изменяет регистр флагов.

Формат команды: `cmp <`***операнд***`1>, <`***операнд***`2>`.

В качестве первого операнда может выступать регистр или значение памяти любого размера (1, 2, 4, 8 байт). В качестве второго операнда может выступать регистр, значение памяти или непосредственное значение. Значение не может быть 64-битным. Одновременно двух значений памяти быть не может. Если 32-битное значение сравнивается с 64-битным, то оно расширяется нулями. В табл. 1.5 приведены команды условных переходов (условия указаны после выполнения команды `cmp x, y`).

Таблица 1.5. Команды условных переходов

Команда	Условие	Условие
JA	X > Y	CF = 0 & ZF = 0
JAE	X >= Y	CF = 0
JB	X < Y	CF = 1
JBE	X < Y	CF = 1 or ZF = 1
JC		CF = 1
JCXZ		CX = 0
JE (то же, что и JZ)	X = Y	ZF = 1
JG	X > Y	ZF = 0 & SF = OF
JGE	X >= Y	SF = OF
JL	X < Y	SF! = OF
JLE	X <= Y	ZF = 1 or SF! = OF
JNA	(X <= Y	CF = 1 or ZF = 1
JNAE	X < Y	CF = 1
JNB	X >= Y	CF = 0
JNBE	X > Y	CF = 1 & ZF = 0
JNC		CF = 0

Таблица 1.5. Команды условных переходов

Команда	Условие	Условие
JNE	X! = Y	ZF = 0
JNG	X <= Y	ZF = 1 or SF! = OF
JNGE	X < Y	SF! = OF
JNL	X >= Y	SF = OF
JNLE	X > Y	ZF = 0 & SF = OF
JNO		OF = 0
JNP		PF = 0
JNS		SF = 0
JNZ	X! = Y	ZF = 0
JO		OF = 1
JP		PF = 1
JPE		PF = 1
JPO		PF = 0
JS		SF = 1
JZ	X = Y	ZF = 1

Иногда имеет смысл использовать команду TEST. Формат этой команды почти такой же, как и у команды CMP, но только в качестве второго операнда не может выступать значение памяти. Эта команда осуществляет операцию «логического И» и изменяет только флаги SF, ZF, PF. Команда TEST полезна для проверки соответствия значения операнда некоторой битовой маске.

1.3.4. Безымянные метки

Компилятор FASM позволяет создавать безымянные метки. *Безымянные метки* – это метки с именем `@@:`.

Для осуществления перехода на безымянную метку нужно указать в качестве метки значения: если нужен переход на ближайшую метку после команды перехода, то следует указать `@f`, а если надо перейти на ближайшую метку до команды перехода, то `@b`. Пример приведён в листинге 1.12.

Листинг 1.12. Использование безымянных меток

```
   ...
@@:
   ;code
   Ja @f     ; переход вперёд
;code
   Jz @b     ; переход назад
   ;code
@@:
   ...
```

В некоторых случаях (например, при реализации сложных алгоритмов) безымянные метки бывают очень полезными, т. к. без них код буквально кишит похожими друг на друга метками.

1.3.5. Работа с битами

Иногда в программе возникает ситуация, когда нужно проверить содержимое некоторого бита либо изменить его состояние. Для этих целей существуют четыре команды работы с битами: BT, BTC, BTR, BTS. Все четыре команды имеют следующий формат: `bt*` <***операнд***>, <***номер бита***>. В качестве операнда может выступать регистр либо переменная в памяти любого размера. Номер бита можно указать как непосредственным значением, так и любым регистром общего назначения. Номера битов нумеруются с нуля.

Команда BT заносит значение указанного бита во флаг CF в регистре флагов. Команда BTC заносит значение указанного бита во флаг CF и производит его инвертирование в операнде. Команда BTR заносит значение указанного бита во флаг CF и обнуляет его в операнде. Команда BTS заносит значение указанного бита во флаг CF и заносит в указанный бит в операнде единицу. Как видно, все указанные команды заносят значение указанного бита во флаг CF, после чего можно использовать команды условного перехода JC и JNC. Пример использования команды BT приведён в листинге 1.13.

Листинг 1.13. Использование команды BT

```
mov ax, 4 ;ax = 100b
bt ax, 2  ; проверка третьего бита в регистре ax
jc metka  ; будет произведён прыжок на метку
....
metka:
....
```

Использование команд BT, BTC, BTR, BTS поможет при оптимизации кода программы; они могут заменить команду TEST для проверки значения конкретного бита.

1.3.6. Резюме

В этом разделе нами было изучено использование данных и меток в программе. Также нами были изучены условные и безусловные переходы. Переходы как условные, так и безусловные являются важнейшими командами ассемблера, т. к. позволяют создавать программы с ветвлением. В разделе 1.4 мы будем более подробно изучать команды ассемблера.

1.4. Изучаем ассемблер подробнее

В предыдущих двух разделах мы изучили основные команды ассемблера. В этом разделе мы изучим более сложные и не менее важные инструкции, и их будет намного больше. Итак, приступим.

1.4.1. Работа с памятью и стеком

Команда XCHG – обмен значений операндов. Синтаксис команды: `xchg <операнд1>, <операнд2>`. В качестве операндов может выступать регистр или значение в памяти. Разумеется, одновременно два значения памяти менять нельзя. Операнды могут быть любого размера.

Команда ADC – сложение с учётом флага переноса. Синтаксис команды полностью идентичен синтаксису команды ADD. Отличие команды ADC от команды ADD заключается только в том, что команда ADC после выполнения сложения прибавляет к результату значение флага CF, т. е. если этот флаг выставлен, то происходит инкремент результата.

Команда LEA – загрузка эффективного адреса. Синтаксис команды: `lea <регистр>, <переменная>`. Команда загружает адрес переменной в регистр общего назначения. В целом без этой команды можно обойтись, но в некоторых случаях она бывает очень полезна. С помощью этой команды можно произвести некоторые вычисления, которые обычно делаются в два этапа; например, после выполнения этой команды `lea eax, [ebx + ebx * 4]` в EAX мы получим значение из EBX, умноженное на 5. Наиболее полезной команда LEA будет в следующем случае: `lea ecx, [edx + 4]`; после выполнения этой команды в регистр ECX будет помещено значение EDX, увеличенное на 4 – таким образом, мы заменяем одной командой LEA пару команд MOV/ADD.

Команда PUSHF – сохранение регистра флагов в стеке. Эта команда сохраняет первые 2 байта регистра флагов в стеке. В защищённом режиме эта команда сохраняет 4 байта регистра флагов в стеке, и она уже называется PUSHFD, но опкод такой же. Соответственно команда PUSHFQ сохраняет 8 байт регистра флагов в стеке.

Команда POPF – «выталкивание» из стека в регистр флагов. Эта команда «выталкивает» из стека значение и заносит его в регистр флагов. Соответственно работают команды POPFD и POPFQ.

Последние две команды созданы для изменения флагов процессора. Зарезервированные флаги изменить нельзя; если программа выполняется на непривилегированном уровне защиты, то привилегированные флаги изменить ей не удастся.

Команда PUSHA – сохранение в стеке регистров общего назначения. Эта команда сохраняет 16-битные регистры общего назначения в стеке. В 64-разрядном режиме команда PUSHA не поддерживается. В защищённом режиме сохраняются 32-битные регистры общего назначения. В защищённом режиме эта команда называется PUSHAD (опкод тот же). Порядок сохранения регистров: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI (в режиме реальных адресов сохраняются их младшие части).

Регистр ESP (SP) сохраняется в том состоянии, в котором он был до выполнения команды, а не в том состоянии, в котором он был после помещения в стек регистра EBX.

Команда POPA – забор из стека в регистров общего назначения. Эта команда забирает из стека регистры общего назначения; 4 позиция в стеке игнорируется, т. к. там находится регистр ESP (SP). В защищённом режиме эта команда называется POPAD (опкод тот же).

Последние две команды работают только в режиме реальных адресов и в защищённом режиме; в 64-битном режиме будет сгенерировано исключение недействительного опкода (#UD).

Команда NOP – пустая: она не принимает параметров и абсолютно ничего не делает. Она только занимает пространство и время. Используется для резервирования места в сегменте кода или организации программной задержки.

1.4.2. Работа с числами на ассемблере

В разделе 1.2 уже упоминалось о том, что числа в процессоре могут быть только целые: знаковые и беззнаковые. Вещественные числа всё равно представляются в двоичном виде как 32-, 64-, 80-битные значения. Оперировать вещественными значениями как таковыми может только математический сопроцессор (при этом нам ничто не мешает вручную реализовать вещественную арифметику и работать с числами с плавающей точкой без использования математического сопроцессора, но это уже совсем другая история).

Беззнаковые числа представляются обычным преобразованием числа в двоичное; следовательно, диапазон значений беззнаковых чисел – $0...2^n-1$, где n – это разрядность числа. У знаковых чисел старший бит обозначает знак числа: если старший бит – единица, то число отрицательное. Все остальные биты – это модуль числа, или инвертированный модуль числа плюс один. Диапазон знакового числа: $-2^{n-1}...2^{n-1} - 1$. На уровне машинных команд между этими двумя видами чисел нет никакой разницы. Находящийся в памяти или в одном из регистров операнд представляет собой (в зависимости от используемой модели адресации) 8-, 16- или 32-разрядное число, все разряды которого абсолютно равноправны. Понятие знака введено исключительно для возможности манипулирования (на логическом уровне) с отрицательными числами – для процессора все числа одинаковы, а для программиста они отличаются тем, что для одних высший разряд выступает в качестве информации о знаке числа (знаковые числа), а для других все разряды несут информации о самом числе (беззнаковые числа). Например, для 16-битного беззнакового числа диапазон значений от 0 до 65535, для 16-битного знакового числа диапазон значений от -32768 до +32767. Двоичное представление беззнакового числа 9B5Ch (39772) будет выглядеть так: 1001101101011100, а если интерпретировать эти биты как число со знаком, то они будут представлять отрицательное число 64A4h (–25764). Принципы сложения и вычитания целых, знаковых и беззнаковых чисел одинаковые, т. е. процессору совершенно безразлично, с какими числами он работает – складывает и вычитает он всегда одинаково, но всё-таки, чтобы программист знал результат операции, процессор устанавливает/сбрасывает флаги переполнения и переноса в зависимости от результата.

При вычислениях очень важными являются флаги переполнения и переноса. В общем случае флаг переноса устанавливается в единицу всякий раз, когда единица выносится за пределы разрядной сетки (при операции сложения) или занимается из этих пределов (при операции вычитания). Флаг переполнения используется как индикатор переполнения при работе с числами со знаком. Он устанавливается в 1, если результат операции над числами со знаком выйдет за пределы допустимого

диапазона результата, и в 0 – в противном случае. Два вышеуказанных флага настолько связаны друг с другом, что становится непонятно, когда именно происходит перенос, а когда переполнение.

Для начала рассмотрим примеры сложения некоторых чисел, результаты сложения и флаги, которые устанавливает процессор после их выполнения (табл. 1.6).

Таблица 1.6. Примеры сложения разных чисел и результаты

№	Операнды	Результат	Флаги	Интерпретация для беззнаковых чисел	Интерпретация для знаковых чисел
1	10000000b 10000000b	00000000b	CF = 1 OF = 1	128 + 128 = 0 неверно	(–127) + (–127) = 0 неверно
2	01000000b 01000000b	10000000b	CF = 0 OF = 1	64 + 64 = 128 верно	64 + 64 = –128 неверно
3	01111111b 00000001b	10000000b	CF = 0 OF = 1	127 + 1 = 128 верно	127 + 1 = –128 неверно
4	11111100b 00000101b	00000001b	CF = 1 OF = 0	252 += 1 неверно	(–4) + 5 = 1 верно
5	01100000b 11100000b	01000000b	CF = 1 OF = 0	96 + 224 = 64 неверно	96 + (–32) = 64 верно
6	11000000b 11000000b	10000000b	CF = 1 OF = 0	192 + 192 = 128 неверно	(–64) + (–64) = –128 верно
7	11111111b 00000001b	00000000b	CF = 1 OF = 0	255 + 1 = 0 неверно	(–1) + 1 = 0 верно

Рассмотрим первый, самый простой случай, когда устанавливаются в единицу оба флага. Итак, при сложении чисел 10000000b и 10000000b (пример № 1) перенос произойдёт только за пределы разрядной сетки, а переноса в седьмой бит (знаковый) не произойдёт. В этом случае устанавливаются в единицу и флаг переноса (CF), и флаг переполнения (OF). Результат сложения как знаковых, так и беззнаковых чисел будет неверен.

Второй случай – это случай, когда устанавливается только флаг переполнения. Так, при сложении чисел 01000000b и 01000000b (пример № 2) или чисел 01111111b и 00000001b (пример № 3) будет выставлен только флаг OF. В обоих приведённых случаях происходит перенос единицы только в знаковый разряд, а переноса за пределы разрядной сетки не происходит. Результат сложения знаковых чисел становится неправильным, результат сложения беззнаковых чисел является правильным.

Следующий случай – это случай, когда устанавливается только флаг переноса. Так, при сложении чисел 11111100b и 00000101b (пример № 4), 01100000b и 11100000b (пример № 5), 11000000b и 11000000b (пример № 6), 11111111b и 00000001b (пример № 7) будет выставлен только флаг переноса. Во всех вышеприведенных примерах происходит как перенос единицы в знаковый разряд, так

и перенос единицы за пределы разрядной сетки. Результат сложения беззнаковых чисел в таких случаях неверен, а результат сложения знаковых чисел – верен.

При выполнении вычитания правила остаются теми же – за исключением того, что происходит не перенос единицы в знаковый разряд или за пределы разрядной сети, а заём из знакового разряда или из-за пределов разрядной сетки.

Если подытожить всё вышесказанное, можно сделать вывод, что при сложении (вычитании) чисел без знака результат будет неверным (с переполнением), если флаг переноса CF равен единице. При сложении (вычитании) чисел со знаком результат будет неверным (с потерей знака), если флаг переполнения OF установлен в единицу. Таким образом, разница между числами со знаком и без знака возникает только на этапе интерпретации их значений или в результате операций с ними.

Очень часто имеет место ситуация, когда необходимо сложить два числа с разрядностью больше чем поддерживает процессор в данном режиме. Например, очень распространены случаи, когда необходимо сложить два 64-разрядных числа в 32-радрядной программе (и вообще в любой программе, работающей в защищённом режиме). Как известно, в защищённом режиме без использования математического сопроцессора нельзя одной командой сложить два 64-разрядных числа. В таких случаях сложение будет происходить в несколько этапов в зависимости от того, какого размера числа необходимо сложить. Сложение 64-разрядных чисел будет происходить в два этапа: сначала сложение младших частей, потом – старших частей с учётом того, произошёл ли перенос единицы в старшую часть (проще говоря, с учётом того, произошло ли переполнение при сложении младших частей). Таким образом, при сложении младших частей надо использовать команду ADD, а при сложении старших частей – команду ADC. Использование команды ADC обусловлено тем, что при сложении младших частей операнды выступают как беззнаковые числа; следовательно, индикатором переполнения будет флаг CF.

В листинге 1.14 приведён пример сложения двух беззнаковых 64-разрядных чисел.

Листинг 1.14. Сложение двух беззнаковых 64-разрядных чисел

```
Val1_high dd 0750AC072h
Val1_low  dd 0780D1E56h

Val2_high dd 0004B015Fh
Val2_low  dd 0E057DAF1h

...
        mov eax, [Val1_low]
        mov edx, [Val1_high]

        add eax, [Val2_low]
        adc edx, [Val2_high]

        jnc @f
        ; произошло переполнение
        @@:
        ; успешное вычисление
```

В результате выполнения кода из листинга 1.4.1 в регистрах EDX:EAX будет содержаться результат сложения двух 64-разрядных чисел. Если происходит сложение чисел со знаком, то после выполнения сложения необходимо проверить содержимое флага OF (команда JNO).

В листинге 1.15 приведён пример сложения двух беззнаковых 64-разрядных чисел, каждое из которых записано в виде одной переменной.

Листинг 1.15. Сложение двух беззнаковых 64-разрядных чисел

```
Val1 dq 05B493E89C0F44Ah
Val2 dq 06A4937796D80D1h
...
        mov eax, dword [Val1]
        mov edx, dword [Val1+4]

        add eax, dword [Val2]
        adc edx, dword [Val2+4]

        jnc @f
        ; произошло переполнение
        @@:
        ; успешное вычисление
```

В случае когда числа представлены в одной переменной, следует помнить, что все числа в памяти хранятся в обратном порядке, т. е. сначала идут младшие байты, а потом старшие.

1.4.3. Умножение и деление

Команда MUL – беззнаковое умножение. Синтаксис команды: `mul <`***операнд***`>`. Команда MUL умножает операнд с регистром EAX(AL, AX, RAX) и сохраняет его в регистрах EDX:EAX (AX, DX:AX, RDX:RAX). Операндом может быть регистр или значение в памяти. В табл. 1.7 представлены возможные результаты работы этой команды.

Таблица 1.7. Зависимость результата работы команды MUL от размера операнда

Размер операнда	Результат
1 байт	AX = AL * <***операнд***>
2 байта	DX:AX = AX * <***операнд*** >
4 байта	EDX:EAX = EAX * <***операнд*** >
8 байт	RDX:RAX = RAX * <***операнд*** >

Команда DIV – беззнаковое деление. Синтаксис команды: `div <`***операнд***`>`. Команда DIV делит регистры EDX:EAX (AX, DX:AX, RDX:RAX) на операнд и сохраняет результат деления в EAX (AL, AX, RAX), а также остаток от деления в EDX (AH, DX, RDX). В табл. 1.8 показаны возможные результаты работы этой команды.

Таблица 1.8. Зависимость результата работы команды DIV от размера операнда

Размер операнда	Источник	Результат деления	Остаток деления
1 байт	AX	AL	AH
2 байта	DX:AX	AX	DX
4 байта	EDX:EAX	EAX	EDX
8 байт	RDX:RAX	RAX	RDX

Команда IMUL – знаковое умножение. Команда может принимать до трёх операндов. У этой команды есть три формы записи в зависимости от того, сколько операндов указано.

1. Первая форма – только один операнд. Команда работает точно так же, как и MUL, только умножение происходит с учётом знака.
2. Вторая форма – два операнда. Синтаксис: `imul` ***<операнд1>***, ***<операнд2>***. В качестве первого операнда может выступать только регистр общего назначения, а вторым операндом может быть регистр, значение памяти или непосредственно значение. Операнды могут быть любого размера, но если второй операнд – непосредственно значение, то оно не может быть размером 8 байт. Первый операнд умножается с учётом знака на второй, и результат сохраняется в первом операнде.
3. Третья форма – это три операнда. Формат: `imul` <***операнд***1>, <***операнд***2>, <***операнд***3>. В качестве первого операнда может выступать только регистр общего назначения. В качестве второго операнда может выступать регистр или значение памяти. Первый и второй операнды должны быть одинакового размера. Третий операнд может быть только непосредственным значением и не может быть размером 8 байт. При использовании этой формы команда умножает с учётом знака второй операнд на третий и результат сохраняет в первом.

Команда IDIV – знаковое деление. Команда IDIV по результату работы полностью идентична команде DIV. Отличие команды IDIV от DIV заключается в том, что если операнды имеют разные знаковые биты, то результат (частное и остаток) будет отрицательным; если же у операндов знаковые биты будут равны, то результат (частное и остаток) будет положительным.

Команды DIV и IDIV не влияют на регистр флагов. Команда MUL устанавливает флаги OF и CF в ноль, если старшая часть результата (AH, DX, EDX, RDX) равна нулю; в любом другом случае эти биты устанавливаются в единицу.

При использовании команды IMUL в первой форме флаги OF и CF устанавливаются в единицу, если какие-либо значащие биты переносятся в старшую часть результата (AH, DX, EDX, RDX), и в ноль, если в старшую часть результата переносов не было. Таким образом, если флаги OF и CF сброшены, то результат может быть считан только из младшей части, и он будет верным.

При использовании команды IMUL во второй и третьей форме флаги OF и CF устанавливаются в единицу, если размерность результата больше, чем размер операнда, указанного в качестве результата. Таким образом, если после операции

умножения выставлены флаги OF и CF, значит, произведение в результирующем операнде неверное (усечённое).

1.4.4. Порты ввода-вывода

Процессор взаимодействует с внешними устройствами через порты ввода-вывода. Порты бывают размером только 1 байт. Два соседних порта формируют один 2-байтовый порт, аналогично 4 соседних порта формируют один 4-байтовый порт. Выводя в порт или считывая из порта значение, устройство реагирует и при необходимости обрабатывает его. Если устройство хочет передать процессору блок данных более чем 4 байта, то оно выводит в порт первую порцию и, как только процессор считал данные, ставит в порт вторую.

Не стоит путать порты ввода-вывода с портами PS/2, USB, IEEE и т. д. Внешнее устройство для процессора (именно для процессора) – это не мышь PS/2, не USB-диск, не CD-ROM; это какой-либо контроллер либо микросхема на материнской плате. Именно через контроллеры и осуществляется взаимодействие процессора с внешними устройствами. Например, взаимодействие с USB-устройствами происходит именно через контроллер шины USB.

Это было небольшое отступление, а теперь перейдем собственно к делу. Всего портов в процессоре 65 535. Для примера: мы выводим в порт с номером 56h значение 1AF4h, в порт 56h – значение 1Ah и в порт 57h – значение F4h. Аналогичная ситуация с 4-байтовыми портами. Для большего быстродействия 4-байтовые порты должны быть кратны четырём, и, соответственно, 2-байтовые порты должны быть кратны двум. Есть четыре команды для работы с портами ввода-вывода: IN, OUT, INS и OUTS. О работе двух последних команд пойдет речь позже.

Команда IN – чтение из порта. Синтаксис команды: `in EAX/AX/AL, <порт>`. Порт может быть задан непосредственно числом или регистром DX. Число может быть не больше 255; если надо получить значение из порта с большим номером, то надо указать номер порта в регистре DX. Размер порта берётся в соответствии с указанным регистром.

Команда OUT – вывод в порт. Синтаксис команды: `out <порт>, EAX/AX/AL`. Порт размером более чем 255 может быть указан только регистром DX.

1.4.5. Циклы

Иногда одно и тоже действие нужно выполнить несколько раз; для этого счётчик повторений можно поместить в некоторый регистр и при каждом повторении уменьшать на единицу этот регистр. Если он равен нулю, то повторять действие не нужно. В листинге 1.16 приведён пример организации цикла таким методом.

Листинг 1.16. Пример организации цикла командами DEC/JNZ

```
    mov ecx, <число повторений>
metka1:
    ...
    <тело цикла>
    ...
    dec ecx
    jnz metka1
```

Последние две строки – это код, который организует цикл. Команда DEC уменьшает значение регистра ECX, и если оно станет равно нулю, то во флаг ZF будет помещена единица; в итоге, если регистр не будет равен нулю, то произойдёт переход на метку.

Но есть более простой метод организации цикла. Команда LOOP принимает в качестве единственного операнда ближнюю метку. Она уменьшает регистр ECX (CX, RCX) на единицу и, если этот регистр не равен нулю, то происходит передача управления метке. Единственное преимущество этой команды состоит лишь в том, что она занимает меньше места.

Также есть две команды организации циклов: LOOPE (LOOPZ) и LOOPNE (LOOPNZ). Команда LOOPE перед передачей управления проверяет флаг ZF и, если он не выставлен, то передачи управления не происходит. Обратное действие производит команда LOOPNE: если флаг ZF выставлен, то передачи управления не происходит.

На современных процессорах организация цикла с помощью команд DEC/JNZ/JZ предпочтительна с точки зрения производительности; в сумме они будут работать в два раза быстрее, нежели одна инструкция LOOP.

1.4.6. Обработка блоков данных

Прежде чем изучать команды работы с блоками данных, мы сначала изучим базовые команды для работы с блоками.

Команда LODSB/W/D/Q – загрузка данных**.** Эта команда загружает байт, слово, двойное слово, четверное слово в регистр AL, AX, EAX, RAX из памяти, на которую указывает регистр ESI (SI, RSI). После этого значение регистра ESI (SI, RSI) увеличивается или уменьшается в зависимости от флага направления DF на 1, 2, 4, 8. Команда не модифицирует регистр флагов. Если DF выставлен, значение уменьшается; если он не выставлен, то увеличивается. Фактически то же самое можно сделать следующими командами:

```
mov ax, word [esi]
add/dec esi, 2
```

Пример приведён для размера данных равного 2 байтам.

Главное преимущество этой и всех нижеследующих команд – меньший размер и возможность их использования со специальным префиксом.

Команда STOSB/W/D/Q – сохранение данных**.** Команда сохраняет байт, слово, двойное слово, четверное слово из регистра AL, AX, EAX, RAX в память, на которую указывает регистр EDI (DI, RDI). Команда не модифицирует регистр флагов. После этого значение регистра EDI (DI, RDI) увеличивается или уменьшается в зависимости от флага направления на 1, 2, 4, 8.

Команда MOVSB/W/D/Q – перемещение данных**.** Команда помещает байт, слово, двойное слово, четверное слово, находящееся в памяти, на которую указывает регистр ESI (SI, RSI), в память по адресу, на который указывает регистр EDI (DI, RDI). Команда не модифицирует регистр флагов. После этого значение регистров EDI (DI, RDI) и EDI (SI, RSI) увеличивается или уменьшается в зависимости от флага направления на 1, 2, 4, 8.

Команда SCASB/W/D/Q – сравнение с памятью**.** Команда сравнивает регистр AL, AX, EAX, RAX со значением в памяти, на которую указывает регистр EDI (DI, RDI), и изменяет соответствующие флаги аналогично команде CMP. После этого значение регистра EDI (DI, RDI) увеличивается или уменьшается в зависимости от флага направления на 1, 2, 4, 8.

Команда CMPSB/W/D/Q – сравнение данных**.** Команда сравнивает байт, слово, двойное слово, четверное слово, находящееся в ячейке памяти, на которую указывает регистр ESI (SI, RSI), и байт, слово, двойное слово, четверное слово, находящееся в памяти, на которую указывает регистр EDI (DI, RDI), и изменяет соответствующие флаги аналогично команде CMP. После этого значение регистров EDI (DI, RDI) и EDI (SI, RSI) увеличивается или уменьшается в зависимости от флага направления на 1, 2, 4, 8.

Команда INSB/W/D – загрузка данных из порта. Команда загружает байт, слово, двойное слово из порта, указанного в регистре DX, в память по адресу, указанному в регистре EDI (DI, RDI). Команда не модифицирует регистр флагов. После этого значение регистра EDI (DI, RDI) увеличивается или уменьшается в зависимости от флага направления на 1, 2, 4.

Команда OUTS/W/D – выгрузка данных в порт. Команда выводит байт, слово, двойное слово в порт, указанный в регистре DX, в память по адресу, указанному в регистре ESI (SI, RSI). Команда не модифицирует регистр флагов. После этого значение регистра ESI (SI, RSI) увеличивается или уменьшается в зависимости от флага направления на 1, 2, 4.

Следует подметить, что во всех командах для получения логического адреса используется сегментный регистр DS вместе с регистром ESI, а с регистром EDI – регистр ES, т. е. в качестве исходного сегмента используется сегмент, описываемый регистром DS, а в качестве сегмента результата – сегмент, описываемый регистром ES.

Теперь перейдем ближе к делу. Все вышеописанные команды полезны только с префиксом REP/REPE/REPNE. Префикс REP уменьшает регистр ECX(CX, RCX) и повторяет команду, если этот регистр не равен нулю.

Листинг 1.17. Пример использования команды STOS

```
metka:
   db 100 dup (0)
...
   mov ecx, 100
   mov edi, metka
   mov al, 0dah
   rep stosb
```

Пример, приведённый в листинге 1.17, заполняет память, определённую меткой `metka`, сотней байтов со значением 0DAh. Далее в листинге 1.18 приведён пример, который копирует память из `metka1` в `metka2`:

Листинг 1.18. Пример использования команды MOVS

```
metka1:
   db 40  dup (11h)
metka2:
   db 40  dup (0)
   mov ecx, 10
   mov esi, metka1
   mov edi, metka2
   rep movsd
```

Префикс REPE (REPZ) делает то же самое, что и rep, но помимо проверки ECX (CX, RCX) на ноль проверяет флаг ZF и, если он выставлен, то происходит повторение. Как уже понятно, префикс REPNE (REPNZ) только повторяет инструкцию, если флаг ZF сброшен.

Листинг 1.19. Использование инструкции SCAS

```
str1:   db "9"
        db 0
str2:   db "qwerty987654"
endstr:
...
   start:
     mov ecx, endstr-str2+1
     std
     mov al, [str1]
     mov edi, endstr-1
     repnz scasb
```

В листинге 1.19 производится поиск символа `str1` в строке `str2`. Значение, оставшееся в регистре ECX, и есть позиция первого символа `str1` с конца в `str2`; если ECX равен нулю, значит, символа нет.

Листинг 1.20. Пример использования инструкции CMPS

```
str1: db "qwerty99",0
endstr1:

str2: db "qwerty98",0
endstr2:
...
   start:
     mov eax, endstr1-str1
     mov ecx, endstr2-str2
     cmp ecx, eax
     jnz _not
 go:
     mov esi, str1
     mov edi, str2
     repz cmpsb
     cmp ecx, 0
```

```
    jz _yes
_not:
    ; строки не равны
_yes:
    ; строки равны
```

В листинге 1.20 сравниваются строки, определённые по соответствующим меткам.

1.4.7. Макросы

Данный материал больше относится к описанию возможностей компилятора FASM, нежели к описанию команд языка ассемблера. Тем не менее без знания возможностей макроязыка FASM программирование на ассемблере намного усложняется. К макросредствам ассемблера относятся константы, макросы и структуры.

Константы можно объявить несколькими способами, но поскольку цель этой книги – не изложение макроособенностей компилятора FASM, то будут описаны только два способа объявления констант.

Первый способ – самый простой: это объявление имени константы и определение её значения через знак равенства. Этим способом можно объявлять константы в любом месте, и они будут доступны в любом месте программы. При использовании константы в программе на её месте будет поставлено её значение.

Второй способ объявления констант – более удобный и гибкий: через директиву EQU. Примеры использования директивы приведены в листинге 1.21.

Листинг 1.21. Использование директив equ и =

```
_C = 45;
_name = 0A1h
Value = "S"; тоже самое что и Value = 53h

CONST1 equ 0123h
CONST2 equ 14d*15h
CONST3 equ "slovo"
CONST4 equ 56-45
CONST5 equ (metka1)
CONST6 equ (metka2+ metka3)
CONST7 equ (CONST2+10b)
CONST8 equ CONST7/2
CONST9 equ (metka1-metka5)
CONST10 equ (metka4+CONST4)
CONST11 equ add edx,edi
```

Из примеров листинга 1.21 видно, что константе можно присвоить значение метки (т. е. её адрес) и даже некоторую команду. Если константа определена как команда, то она не может использоваться как операнд; её можно указать просто одиночно, и при этом она будет заменена своим значением. Вообще во всех случаях определения констант при применении константы она будет заменена своим значением. Тем не менее у директив = и equ есть два существенных отличия.

Листинг 1.22. Использование директив equ и =

```
C1 = 10
C2 equ 2
C3 equ C1+C2
C4 = C1+C2

mov eax, C3 ; mov eax, C1+C2
mov eax, C4 ; move ax, 12
```

После анализа вышеприведённого примера становится понятно, что при использовании директивы = значение константы вычисляется сразу и при подстановке вставляется вычисленное значение. А при использовании equ значение константы не вычисляется и просто заменяется её определением при использовании. Таким образом, подстановочное значение константы, объявленной с помощью директивы equ, может изменяться.

С константами мы разобрались. Теперь поговорим о макросах. Макрос – это более «продвинутый» вариант директивы equ. *Макрос* – это набор инструкций. Макросы объявляются следующим образом:

```
macro <имя макроса> <параметры макроса>
{
<тело макроса>
}
```

Потом при каждом упоминании имени макроса оно будет заменено телом макроса, где бы он ни использовался. Вот один из примеров макроса:

```
macro tst {cmp eax,ebx}
```

Вы можете использовать инструкцию TST в любом месте после её определения, и она будет ассемблирована как cmp eax,ebx. Определение константы с таким значением даст похожий результат; различие лишь в том, что имя макроинструкции будет распознаваться только как мнемоник инструкции, т. е. как команда ассемблера. Также макроинструкции заменяются соответствующим кодом даже перед заменой символьных констант их значениями. Т. е., если вы определите макроинструкцию и константу под одним и тем же именем и используете это имя как мнемоник инструкции, оно будет заменено содержанием макроинструкции, но если вы используете его внутри операндов, имя будет заменено значением константы.

Листинг 1.23. Пример макроса с тремя параметрами

```
macro mov op1,op2,op3
{
if op3 eq
mov  op1,op2
else
mov  op1,op2
mov  op2,op3
end if
}
```

Теперь поподробнее проанализируем макрос, приведённый в листинге 1.23. Сначала пара слов о директиве `if`. Эта директива производит условное ассемблирование. Если условие выполняется, то ассемблируется блок инструкций после неё до следующей команды `end if`, `else` или `elseif`. Директива `else` ассемблирует блок инструкций, если не были выполнены все предыдущие условия. Директива `elseif` ассемблирует блок инструкций, если будет выполнено условие и если не будут выполнены все предыдущие условия. Директива `end if` заканчивает всю цепь инструкций `if`. Директива `eq` проверяет операнды на тождественность; они могут быть записаны в разных формах. В нашем случае она проверяет, определена ли она вообще, и если определена, то производится действие op1 = op2, op2 = op3. После этого при использовании

```
mov eax, ebx, edx
```

произойдет замена на

```
mov eax, ebx
mov ebx, edx
```

Если будут указаны только два операнда, то произойдет замена на обычную команду `mov`.

Описание макросного движка компилятора FASM требует отдельного разговора. Однако приведенной здесь базовой информации о макросах должно хватить для понимания примеров из этой книги.

1.4.8. Структуры

Структура – это набор переменных (данных). Структура задаётся с помощью директивы STRUC. Пример объявления структуры приведён в листинге 1.24.

Листинг 1.24. Пример объявления структуры

```
Struc sample
{
   .x1 dd ?
   .y1 dw ?
   .y2 db ?
}
```

Теперь после объявления структуры можно определить переменную и использовать её поля:

```
...
_Var sample
...
mov ax, [_Var.y1]
```

При использовании полей структуры выражение заменяется адресом равным смещению переменной плюс смещение поля в самой структуре.

```
...
_Var sample
...
mov ax, [_Var.y1]
```

Аналогичная запись:

```
mov ax, word [_Var+4]
```

Если требуется задавать значения полей и эти значения заранее неизвестны, то можно воспользоваться параметрами, как при использовании макросов (листинг 1.25).

Листинг 1.25. Объявление структуры с параметрами

```
struc numbers x,y
{
   .x1 dd x
   .y1 dw ?
   .y2 db y
}
```

После этого, объявляя переменную, надо обязательно указать все параметры:

```
...
_Var numbers 34, 67
...
mov ax, [_Var.y1]
```

Чтобы можно было воспользоваться структурами, адресованными регистрами, следует применить директиву `virtual` (листинг 1.26).

Листинг 1.26. Использование директивы VIRTUAL

```
...
_Var numbers
...

mov  ebx, _Var

virtual at ebx
.ebx numbers
end virtual
mov  dl, [.ebx.y2]
```

Использование префикса `.ebx` является вынужденной мерой, т. к. если написать просто `ebx` внутри директивы `virtual`, компилятор выдаст ошибку, потому что имя регистра нельзя использовать в качестве имени метки.

1.4.9. Работа с MSR-регистрами

Регистры MSR (Model Specific Registers) – это регистры, специфичные для каждого типа машин. Одни и те же регистры на одном типе процессоров могут отвечать за одно, а на другом типе процессоров – уже совсем за другое. Каждый MSR-регистр имеет свой индекс, по которому к нему можно обратиться. Все MSR-регистры 64-битные. Для работы с регистрами MSR существует две привилегированные команды: RDMSR, WRMSR.

Команда RDMSR сохраняет содержимое MSR-регистра, индекс которого указан в ECX(RCX), в регистры EDX:EAX (RDX:RAX). Если мы находимся в 64-битном режиме, то старшие части регистров RDX и RAX будут очищены. Команда WRMSR загружает содержимое регистров EDX:EAX (RDX:RAX) в MSR-регистр,

индекс которого указан в ECX (RCX). Если мы находимся в 64-битном режиме, то значения будут браться из младших частей регистров RDX и RAX.

Эти две команды были введены в процессорах Pentium I. Как уже было сказано, эти команды привилегированные, т. е. их можно вызвать только на нулевом уровне привилегий. Также их можно вызвать в режиме реальных адресов.

1.4.10. Команда CPUID

Команда CPUID используется для идентификации процессора. С помощью этой команды можно узнать, на каком типе процессора работает наша программа, какие технологии поддерживает процессор.

Команда CPUID возвращает в регистры EAX, EBX, ECX, EDX информацию в зависимости от того, какой тип информации был указан в регистре EAX (иногда ещё и в регистре ECX). Например, если мы вызовем команду CPUID и в регистре EAX будет 0, то в регистр EAX будет сохранено максимальное значение, которое можно положить в регистр EAX для вызова команды CPUID.

В качестве примера можно привести кусок кода, который получает название процессора. После выполнения кода, приведённого в листинге 1.27, в строке `message` будет содержаться строка с названием процессора.

Листинг 1.27. Пример использования команды CPUID

```
   message db  49 dup (0)
...
      mov edi, message
      mov eax, 80000002h
      cpuid
      mov [edi], eax
      add edi, 4
      mov [edi], ebx
      add edi, 4
      mov [edi], ecx
      add edi, 4
      mov [edi], edx
      add edi, 4
      mov eax, 80000003h
      cpuid
      mov [edi], eax
      add edi, 4
      mov [edi], ebx
      add edi, 4
      mov [edi], ecx
      add edi, 4
      mov [edi], edx
      add edi, 4
      mov eax, 80000004h
      cpuid
      mov [edi], eax
      add edi, 4
```

```
mov [edi], ebx
add edi, 4
mov [edi], ecx
add edi, 4
mov [edi], edx
```

Если указан тип информации 80000002h, то мы получим первые 16 байт строки названия процессора. Эта строка будет содержаться в регистрах EAX, EBX, ECX, EDX в указанном порядке. Для получения следующих 16 байт строки названия процессора надо вызвать команду CPUID, указав тип информации равный 80000003h. Для получения последних 16 байт строки названия процессора надо указать тип информации 80000004h.

Более подробно получение характеристик процессора будет описано в следующих разделах, когда мы начнем изучать механизмы процессора и понадобится узнать, поддерживает ли процессор некоторый механизм или технологию, необходимую для дальнейшей работы программы.

1.4.11. Команда UD2

Напоследок расскажем немного о команде UD2. Команда UD2 – это фактически не команда, а просто неверный опкод. Она специально была создана для генерации исключения неверного опкода. Исключение неверного опкода может быть сгенерировано при выполнении любой команды, которая не поддерживается процессором; таких команд может быть сколько угодно, и притом неизвестно, на каком процессоре будет выполняться программа. Команда UD2 гарантированно генерирует исключение неверного опкода.

1.4.12. Включение файлов

Иногда нужно использовать одни и те же структуры и макросы в разных программах. Определять их в каждой программе очень долго и неудобно. Для этого можно определить все часто используемые макросы в отдельном файле и просто включать их в свои программы. Директива `include` производит включение текстового файла в текст вашей программы. После неё надо указать путь к включаемому файлу в кавычках. Например: `include 'file.ext'`. Файл может быть с любым расширением (.txt, .inc, .asm и т. д.) – главное, чтобы он был текстового формата. После этого можно считать, что вместо директивы `include` подставлено всё содержимое этого файла.

Включить файл можно в любом месте программы, и именно в этом месте будет находиться содержимое включаемого файла. Можно использовать идентификаторы (константы, метки, макросы и т. д.), объявленные во включённом файле в основной программе без каких-либо ограничений.

1.4.13. Резюме

Вот мы и подошли к концу первой главы. В ней мы изучили основные команды ассемблера и получили фундамент для изучения программирования в различных режимах процессора. В начале следующей главы (раздел 2.1) речь пойдёт о защищённом режиме процессора. Начиная с этого момента мы подразумеваем, что читатель имеет средний уровень знаний и знает команды, используемые в примерах.

2 Защищённый режим

2.1. Введение в защищённый режим

В предыдущей главе мы изучили основы ассемблера. Теперь, получив необходимые базовые знания, мы можем спокойно приступить к изучению защищённого режима. В этой главе будут рассмотрены принципы и механизмы работы процессора в защищённом режиме. Конкретно в данном разделе пойдёт речь об общей характеристике защищённого режима. В конце раздела для закрепления полученных знаний мы напишем приложение, которое переводит процессор в защищённый режим.

Защищённый режим называется так потому, что он позволяет защитить данные операционной системы от приложений, работающих в ней, а также защитить данные приложений друг от друга. Защита достигается благодаря разделению приложений на разные уровни привилегий.

После включения или аппаратного сброса (перезагрузки) процессор находится в режиме реальных адресов. Это и есть 16-битный режим работы процессора. В этом режиме используется 20-битная система адресации, с помощью которой можно адресовать только 1 Мб физической памяти. В режиме реальных адресов работает операционная система MS-DOS. Теперь забудем о режиме реальных адресов и сконцентрируемся на защищённом режиме, который даёт нам возможность использовать ресурсы компьютера (память, производительность, защита и др.) на полную мощность.

2.1.1. Уровни привилегий

Режим реальных адресов и защищённый режим – это два совершенно разных режима. При работе в защищённом режиме процессор следит за правильным выполнением текущей программой ряда условий: например, она не должна выполнять некоторые инструкции или обращаться к некоторым областям памяти. Если всё же происходит нарушение какого-либо условия, то процессор генерирует специальный тип прерывания – так называемое исключение, и снабжает это прерывание информацией, описывающей, где произошло нарушение и как оно произошло. Затем специальная процедура (обработчик прерывания) обрабатывает это прерывание и решает, что дальше делать с программой (например, прекратить её выполнение).

Определением условий работы и ограничений для программ должна заниматься операционная система. Когда программа переводит процессор в защищённый режим, то становится полноправным хозяином компьютера – по сути, операционной системой. Она устанавливает систему разрешений и условий для других программ. Для того чтобы эти условия и разрешения не смогла переопределить

другая программа, в процессоре введена система уровней привилегий. В режиме реальных адресов нет системы привилегий, т. к. там может выполняться только одна программа, поэтому данные защищать не от кого (за исключением некоторых случаев). Благодаря тому что в защищённом режиме можно задействовать механизм многозадачности, на процессоре может выполняться одновременно несколько задач (вернее, одновременно они не выполняются, а просто быстро сменяют друг друга). Поэтому помимо самой операционной системы на процессоре может выполняться ещё несколько задач, и каждая может нарушить целостность данных и подвергнуть данные пользователя опасности; именно поэтому и введена система привилегий. Благодаря системе привилегий прикладная программа не может изменить правила, которые установила операционная система, и тем самым получить доступ к важным данным. Благодаря этому операционная система, например, может разрешить работу с дисковыми накопителями только для себя – и тогда вирусы, которые будут работать напрямую с дисковыми накопителями через порты ввода-вывода, окажутся бессильными.

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 0 – самый привилегированный. Когда программа работает на этом уровне привилегий, ей «можно всё». Уровень 1 – менее привилегированный, и запреты, установленные на уровне 0, действуют для уровня 1. Уровень 2 – ещё менее привилегированный, а 3-й имеет самый низкий приоритет. Таким образом, оптимальная (классическая) схема работы программ по уровням привилегий будет следующая:

- уровень 0: ядро операционной системы;
- уровень 1: драйверы ОС;
- уровень 2: интерфейс ОС;
- уровень 3: прикладные программы.

Разумеется, это не единственный способ. Можно, например, не отделять драйверы и компоненты, отвечающие за интерфейс, от ядра. Тогда в такой операционной системе не будет реализован встроенный в процессор механизм защиты программ и данных друг от друга. В результате такая операционная система будет неустойчивой к сбоям внутри компонентов, работающих на уровне ядра системы. К примеру, точно такой принцип разделения компонентов применяется в операционных системах Windows; в случае сбоя в работе какого-либо драйвера перестаёт работать вся система в целом.

Уровни привилегий 1, 2 и 3 подчиняются условиям, установленным на уровне 0, поэтому функционально эти четыре уровня можно разделить на две 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. Тем не менее самые популярные операционные системы (UNIX и Windows) используют только два уровня привилегий – 0 и 3, наверное потому, что так проще.

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 в отношении нарушителя.

Сам по себе уровень привилегий ещё ничего не значит, его нельзя «установить в процессоре». Уровень привилегий применяется как одно из свойств при описании различных объектов, например сегмента кода, и действует при работе только с этим объектом.

2.1.2. Сегменты в защищённом режиме

В защищённом режиме можно использовать до 4 Гб физической памяти в стандартном режиме и 64 Гб в специальном (подробнее о нём мы расскажем в следующих разделах). Но прежде чем начать описывать принципы работы памяти в защищённом режиме, надо ещё раз вспомнить устройство памяти в реальном режиме. Вся память делится на сегменты размером 64 Кб; для получения доступа к памяти надо указать номер сегмента и смещение в этом сегменте. Физический адрес, который выставляется на шину адреса, вычисляется таким образом:

```
физический_адрес = сегмент * 10h + смещение
```

Смещение указывается 16-битным значением в регистре общего назначения или непосредственным значением. Сегмент тоже указывается 16-битным значением в сегментном регистре.

Адрес памяти в защищённом режиме является 32-битным, следовательно, можно адресовать 4 Гб физической памяти. Вся память делится на сегменты. Каждый сегмент может быть размером от 0 до 4 Гб (т. е. покрывать всю память). Для доступа к любому адресу в памяти, т. е. к любому сегменту и любому смещению, в нём надо сформировать логический адрес. Логический адрес представляет из себя описание сегмента и адрес в сегменте. В реальном режиме сегмент указывался просто его порядковым номером. В защищённом режиме сегмент указывается селектором. *Селектор сегмента* – это его идентификатор. Селектор имеет размер 16 бит, точно так же как и в режиме реальных адресов, но обозначает он совсем другое. Впрочем, о нём поговорим попозже.

Для того чтобы описать сегмент, надо указать его начальный адрес, уровень привилегий для доступа к нему, его лимит (т. е. его размер) и т. д. Для описания сегмента памяти отводится специальная структура размером 8 байт. Эта структура называется *дескриптором*. Если быть более точным, то любой объект, которым управляет процессор и к которому он регулирует доступ, описывается дескриптором. Все дескрипторы объединяются в специальных таблицах дескрипторов – глобальной и локальной, но об этом опять же поговорим позднее. Каждый объект описывается своим типом дескриптора. Сегмент памяти описывается сегментным дескриптором.

Рис. 2.1. Общий формат дескриптора

Сегментный дескриптор принято делить на два двойных слова. На рис. 2.1 приведён общий формат дескриптора.

Базовый адрес – 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. Сегмент может измеряться в байтах или страницах. Если он измеряется в байтах, то размер сегмента может быть максимум 1 Мб; если в страницах, то 4 Гб.

Тип сегмента – 4-битовое поле, определяющее тип сегмента. Каждый бит типа сегмента имеет следующие значения:

1. Старший бит (бит 11 во втором двойном слове): если 0, то это сегмент данных, если 1 – то кода. Разумеется, если данный бит сброшен, то передача управления коду, который находится в памяти, описываемой этим дескриптором, будет невозможна.
2. Бит 10 во втором двойном слове. Если это сегмент кода, то он показывает подчинённость этого сегмента (более подробно об этом будет рассказано в разделе 2.5). Если это сегмент данных, то данный бит показывает направление расширения сегмента: если 0, то вверх (в сторону старших адресов) – как обычно; если 1, то вниз (в сторону младших адресов) – как в стеке.
3. Бит 9 во втором двойном слове. Если это сегмент кода, то данный бит говорит нам, можно ли читать из этого сегмента (если 0, то только выполнение, если 1, то выполнение и чтение). Если это сегмент данных, то данный бит показывает, можно ли записывать в этот сегмент (если 0, то только чтение, если 1, то чтение и запись).
4. Бит 8 во втором двойном слове – A (accessed). Этот бит показывает, был ли произведен доступ к сегменту, описываемому этим дескриптором. Если процессор обращался к сегменту для чтения или записи данных или для выполнения кода, размещённого в нём, то бит A будет установлен (равен 1), в противном случае – сброшен (0). С помощью бита A операционная система может определить, использовался ли за последнее время этот сегмент 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).

Бит S (System). Если этот бит установлен, то дескриптор определяет сегмент кода или данных, а если сброшен, то системный объект (о системных объектах мы будет говорить в других разделах). В сегментном дескрипторе этот бит установлен всегда – иначе четыре предыдущих флага интерпретировались бы совсем подругому.

Поле DPL (Descriptor Privilege Level). Уровень привилегий, который имеет объект, описываемый данным дескриптором. Это 2-битовое поле, в которое при создании дескриптора записывают значения от 0 до 3, определяющие уровень привилегий. Например, у кода, который выполняется на нулевом уровне привилегий, в дескрипторе сегмента кода оба бита должны быть сброшены. Именно по этим битам (и по уровню привилегий сегмента стека) процессор и определяет уровень привилегий текущего выполняемого кода.

Бит P (Present). 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 отсутствующего сегмента и запустит обработчик этой ситуации, который должен будет подгрузить содержимое сегмента с диска и установить бит P. Затем управление снова передаётся команде, обратившейся к данному сегменту (производится повторное выполнение команды, вызвавшей сбой), и работа программы продолжается. Бит P устанавливается и сбрасывается программами, сам процессор его только считывает. Но в реальности механизм подкачки осуществляется через механизм страничного преобразования. Обычно этот бит используется в случаях, когда нужно пометить свободный элемент в дескрипторной таблице.

Бит AVL. Этот бит процессор не использует и позволяет программе или операционной системе использовать его в своих целях.

Бит L (только 64-битный режим). В защищённом режиме этот бит зарезервирован и всегда сброшен. Если процессор находится в 64-битном режиме и если бит сброшен, то процессор работает в режиме совместимости. Если этот бит выставлен, то бит D должен быть сброшен. Подробнее об этом бите пойдет речь в главе 4.

Бит D (Default size). Если бит сброшен, по процессор использует объект, описываемый данным дескриптором, как 16-разрядный; если 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). Если в регистре SS содержится селектор дескриптора, у которого выставлен данный бит, то размер элемента в стеке 32-битный, а если этот флаг сброшен, то 16-битный. В защищённом режиме допускается использование одновременно 16- и 32-разрядных сегментов, но при написании новых программ подразумевается, что все сегменты будут 32-разрядные. Вряд ли когда-либо вам пригодится сбрасывать этот бит, разве что в 64-битном режиме, ибо одновременно биты D и L не могут быть выставлены.

Бит G (Granularity). Если бит G=0, то сегмент имеет байтную гранулярность, иначе – страничную (одна страница – это 4 Кб). Например, сегмент, имеющий

предел, равный 0FFh, при G = 0 будет иметь размер 255 байт, а при G = 1 – 1020 Кб (255 страниц).

С дескриптором сегмента мы разобрались. Все дескрипторы хранятся в глобальной и локальной дескрипторной таблице (подробнее о локальной дескрипторной таблице будет идти речь в следующих разделах).

2.1.3. Глобальная дескрипторная таблица

Глобальная дескрипторная таблица (далее GDT) хранится в памяти. Глобальную дескрипторную таблицу можно расположить по любому адресу в памяти. Рекомендуется располагать её по адресу, выровненному на границу 8 байт (т. е. кратному 8), поскольку при этом увеличивается скорость доступа к её элементам (дескрипторам). В GDT может храниться до 8192 дескрипторов. Нулевой дескриптор, т. е. дескриптор, определённый в самом начале GDT, процессор не использует. Если всё же в программе встречается обращение к нулевому дескриптору, то процессор генерирует исключение и не позволит доступ к такому дескриптору, хотя при загрузке в сегментный регистр селектора нулевого дескриптора исключение не будет генерироваться. Адрес таблицы и её лимит хранятся в регистре GDTR. Лимит таблицы нужен для того, чтобы не произошло обращения за границу таблицы. Регистр GDTR в защищённом режиме имеет размер 6 байт; его формат приведён на рис. 2.2.

Таблица может быть не больше чем 64 Кб, отсюда максимальное количество дескрипторов 8192 (65536/8=8192), точнее 8191, т. к. нулевой дескриптор не используется.

Для загрузки регистра GDTR используется команда LGDT. Эта команда является привилегированной и может выполняться программой, которая находится на нулевом уровне привилегий. Операндом должны быть 6 байт памяти. Для получения содержимого GDTR используется команда sgdt, формат команды такой же. Но она уже может выполняться на любом уровне.

Теперь вернёмся к началу. Что такое селектор? Селектор – это индекс дескриптора в GDT. Селектор имеет размер 2 байта и содержится в сегментом регистре или задаётся непосредственным значением. Формат селектора приведён на рис. 2.3.

Бит TI указывает, к какой таблице следует обратиться. Подробнее этот бит будет рассматриваться в следующих разделах (в частности, 2.4). Поле RPL обозначает запрашиваемый уровень привилегий.

Теперь о преобразовании логического адреса. Обратиться к памяти можно несколькими способами:

- сегмент указывается сегментным регистром, адрес – непосредственным значением или регистром;

Рис. 2.2. Формат регистра GDTR

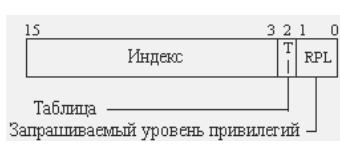


Рис. 2.3. Формат селектора

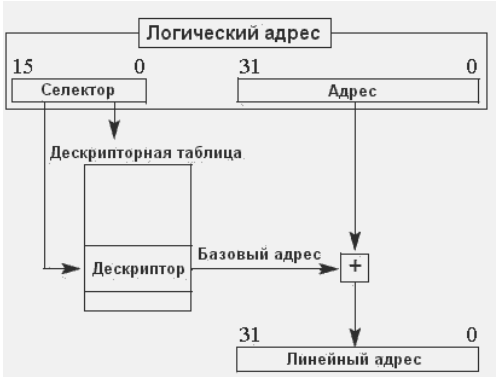


Рис. 2.4. Преобразование адреса в защищённом режиме

- сегмент указывается непосредственным значением селектора, адрес – непосредственным значением или регистром. В этом случае непосредственное значение загружается в соответствующий сегментный регистр (при прыжках селектор загружается в регистр CS, а при обращении к памяти – в DS);
- просто смещение. В этом случае селектор берётся из соответствующего сегментного регистра (при прыжках из CS, при работе с данными – из DS).

Как видите, значение селектора будет использовано в любом случае. Поэтому преобразование логического адреса в линейный будет происходить при каждом обращении к памяти. Схема преобразования логического адреса в линейный изображена на рис. 2.4.

Из селектора извлекается поле индекса, по индексу находится соответствующий дескриптор, из дескриптора извлекается поле адреса базы. К адресу базы добавляется смещение. В итоге получается линейный адрес. Линейный адрес преобразуется в физический. В наших программах мы пока не будем использовать механизм трансляции страниц (о котором пойдет речь в следующих разделах), поэтому логический адрес у нас всегда будет равен физическому адресу.

При каждом обращении к памяти происходит преобразование адреса. Если программа попытается использовать неверный дескриптор или дескриптор высокого уровня привилегий, то процессор сгенерирует соответствующее исключение. Для увеличения производительности у каждого сегментного регистра есть теневая часть, доступная только процессору, размером 8 байт, и при каждой загрузке селектора в регистр процессор проверяет установки защиты – если всё прошло успешно, то он загружает весь дескриптор в теневую часть сегментного регистра. В дальнейшем при каждом обращении к памяти процессор обращается не к таблице, а к теневой части сегментного регистра. И если изменить в таблице её используемый дескриптор, то никаких изменений не произойдёт, т. к. надо еще раз загрузить требуемый селектор в соответствующий сегментный регистр.

2.1.4. Практика

Итак, основы работы в защищённом режиме процессора мы рассмотрели, а именно: для того чтобы обратиться к любому байту памяти, надо указать (явно или неявно) селектор сегмента (не номер сегмента в памяти) и смещение в сегменте; сегмент описывается в дескрипторе. Все дескрипторы (пока мы знаем только дескрипторы сегментов) находятся в глобальной дескрипторной таблице. Базовый адрес глобальной дескрипторной таблицы (начальный адрес или адрес первого дескриптора) находится в регистре GDTR. Первый дескриптор не используется.

После включения процессор находится в режиме реальных адресов. Напишем программу, которая переводит процессор из режима реальных адресов в защищённый режим.

За перевод процессора в защищённый режим отвечает всего лишь один бит системного регистра CR0, а именно нулевой. Нижеприведённый код переводит процессор в защищённый режим:

```
mov             eax,cr0
or              al,1
mov             cr0,eax
```

Но не так всё просто, как хотелось бы. Если перевести процессор в защищённый режим, не подготовив процессор к переходу, то после перехода процессор сразу «зависнет» (если модель старая) или перезагрузится. Во-первых, надо определить глобальную дескрипторную таблицу, т. к. в сегментном регистре CS будет находиться номер сегмента кода режима реальных адресов, а он уже будет интерпретироваться как селектор сегмента кода. Во-вторых, после выполнения инструкции `mov cr0,eax` из-за неверного значения в регистре CS и отсутствия глобальной дескрипторной таблицы будет сгенерировано исключение общей защиты, а поскольку у нас не определена глобальная таблица прерываний, будет сгенерировано исключение двойной ошибки. После того как не будет найден обработчик исключения двойной ошибки (ведь глобальная таблица прерываний не определена), процессор перезагрузится. Подробнее о глобальной таблице прерываний и об исключениях пойдёт речь в следующем разделе.

Также надо обновить EIP на смещение в новом сегменте кода, который мы описали в GDT. Следовательно, первая инструкция защищённого режима должна быть инструкцией загрузки в регистр cs селектора сегмента кода и обновления EIP. Это должна быть инструкция дальней передачи управления. Единственная инструкция, которая может выполнить загрузку непосредственного значения в регистр CS, это команда

`jmp <`***селектор***`>:<`***смещение***`>`

Но почему именно JMP, почему нельзя использовать другие команды для перехода CALL или RET? Потому что команда CALL использует стек для сохранения адреса следующей инструкции, а поскольку в сегментном регистре стека у нас стоит всё ещё старый номер сегмента стека, будет сгенерировано исключение общей защиты и, конечно же, процессор «зависнет». Команда RET вообще не подходит, т. к. для её использования адрес перехода должен содержаться в стеке, а стек у нас не определён.

Также перед переходом в защищённый режим надо запретить все прерывания. Ведь первое же срабатывание прерывания таймера «подвесит» наш процессор, потому что таблица прерываний не определена! (О прерываниях будет идти речь в следующем разделе.)

Далее в листинге 2.1 приведён код программы, которая переводит процессор в защищённый режим.

Листинг 2.1. Программа, переводящая процессор в защищённый режим

```
ORG 100h

start:
; очистка экрана:
     mov    ax,3
     int    10h

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

; вычисляем линейный адрес точки входа в защищенный режим
     xor    eax,eax
     mov    ax,cs
     shl    eax, 4
     add    eax, PROTECTED_MODE_ENTRY_POINT
     mov    [ENTRY_OFF],eax

; теперь надо вычислить линейный адрес GDT
     xor    eax,eax
     mov    ax,cs
     shl    eax,4
     add    ax, GDT

; линейный адрес GDT кладем в заранее подготовленную переменную:
     mov    dword [GDTR+2],EAX

; загрузка регистра GDTR:
     lgdt   fword [GDTR]

; запрет всех прерываний:
     cli

     in   al,70h
     or   al,80h
     out    70h,al

; переключение в защищенный режим:
     mov    eax,cr0
```

```
      or   al,1
      mov    cr0,eax

; загрузить новый селектор в регистр CS
      db 66h  ; префикс изменения разрядности операнда
      db 0EAh ; опкод команды JMP FAR
ENTRY_OFF dd PROTECTED_MODE_ENTRY_POINT; 32-битное смещение
      dw 00001000b ; селектор первого дескриптора
; ГЛОБАЛЬНАЯ ТАБЛИЦА ДЕСКРИПТОРОВ
GDT:
NULL_descr db  8 dup(0)
CODE_descr db  0FFh,0FFh,00h,00h,00h,10011010b,11001111b,00h
DATA_descr db  0FFh,0FFh,00h,00h,00h,10010010b,11001111b,00h
VIDEO_descr db 0FFh,0FFh,00h,80h,0Bh,10010010b,01000000b,00h
GDT_size  equ  $-GDT

label GDTR fword
      dw GDT_size-1; 16-битный лимит GDT
      dd ? ; здесь будет 32-битный линейный адрес GDT

 use32 ; далее следует 32-битный код

PROTECTED_MODE_ENTRY_POINT:
; загрузим сегментные регистры требуемыми селекторами
      mov    ax,00010000b
      mov    bx,ds     ; номер сегмента кода режима реальных адресов
      mov    ds,ax
      mov    ax,00011000b
      mov    es,ax

      xor    esi,esi
      mov    si,bx
      shl    esi,4
      add    esi, message
      xor    edi,edi
      mov    ecx,18

; вывод на экран:
      rep    movsb
      jmp    $
message: db '1',35h,'2',35h,'3',35h,'4',35h,'5',35h,'6',35h,'7',35h,'8',
35h,'9',35h
```

Итак, всё по порядку. Назначение директивы `ORG` будет объяснено чуть позже – для нас важно то, что директива `ORG 100h` сообщает компилятору FASM, что необходимо генерировать COM-программу. COM-программы являются простым типом исполняемых файлов в DOS. COM-программу по другому можно называть «куском памяти в файле», при загрузке операционная система DOS находит свободный сегмент данных и просто копирует файл в память со смещением 100h

(256 байт) относительно начала сегмента. Первые 256 байт необходимы для хранения служебной информации. Как видно, COM-программы не могут занимать более одного сегмента в памяти и имеют максимальный размер 64 Кб (для наших целей этого более чем достаточно).

Теперь о структуре GDT. В глобальной дескрипторной таблице будет три дескриптора для трёх сегментов: сегментов кода, данных и так называемого сегмента видеопамяти. Сегмент данных и кода у нас будет покрывать всё линейное адресное пространство. Сегмент видеопамяти будет начинаться с адреса 0B8000h и иметь лимит 0FFFFh. Первые две инструкции выполняют очистку экрана в текстовом режиме (именно в этом режиме находится видеокарта после перезагрузки), для этого мы вызываем сервис DOS. Следующие три инструкции выполняют включение 32-битной адресации. Поскольку в режиме реальных адресов используется 20-битная адресация, попытка записи по адресу 100000h приведет к записи по адресу 0h. Для этого надо установить второй бит в порте ввода/вывода по адресу 92h.

Следующие две группы инструкций выполняют вычисление линейных адресов точки входа в защищённый режим и глобальной таблицы дескрипторов. Это делается потому, что компилятор во время компиляции выставит смещения всех меток как смещения в текущем сегменте, т. к. он компилирует программу по правилам DOS. Вычисление производится по известной формуле:

физический_адрес = ***сегмент*** * 10h + ***смещение***

Умножение на 10h (16 или 2^4) – это то же самое, что и сдвиг влево на 4 бита. Вычисленные адреса сохраняются в соответствующие переменные.

Далее команда

```
lgdt            fword [GDTR]
```

загружает регистр GDTR. Потом происходит запрещение всех прерываний и переключение в защищённый режим. Первую команду защищённого режима мы записали в виде её опкода. Адрес прыжка мы вычислили во время подготовки.

Далее идёт загрузка сегментных регистров (регистра сегмента данных и регистра сегмента работы со строками).

Потом идёт код, который выводит строку на экран. Для этого просто копируется строка в видеопамять. В видеопамяти каждые 2 байта интерпретируются как символ и формат вывода этого символа. Первый байт – это сам символ, а второй байт – цвет символа и цвет фона. Цвет 4-битный, т. е. максимальное количество цветов – 16. Старшие 4 бита обозначают цвет фона, а младшие 4 бита – это цвет символа. Так, запись 0F означает белый символ на чёрном фоне. Вычисление адреса источника происходит всё по той же причине: при компиляции FASM подставил вместо меток адреса по правилам DOS, т. е. смещения в сегменте.

После выполнения всех действий программа зацикливает процессор – здесь поможет только перезагрузка.

Всегда вычислять линейные адреса по правилам режима реальных адресов неудобно. Для максимального удобства немного переделаем код нашей программы (листинг 2.2).

Листинг 2.2. Программа, переводящая процессор в защищённый режим

```
ORG 100h

     STACK_BASE_ADDRESS    equ 200000h
     USER_PM_CODE_BASE_ADDRESS  equ 400000h
     USER_PM_CODE_SIZE equ USER_PM_CODE_END - USER_PM_CODE_BASE_ADDRESS

     CODE_SELEKTOR  equ 8h
     DATA_SELEKTOR  equ 10h
     VIDEO_SELEKTOR equ 18h

start:
     mov    ax,3
     int    10h

     in   al,92h
     or   al,2
     out    92h,al

     xor    eax,eax
     mov    ax, cs
     shl    eax,4
     add    eax, PROTECTED_MODE_ENTRY_POINT
     mov    [ENTRY_OFF],eax

     xor    eax,eax
     mov    ax,cs
     shl    eax,4
     add    ax, GDT

     mov    dword [GDTR+2],eax
     lgdt   fword [GDTR]

     cli
     in   al,70h
     or   al,80h
     out    70h,al

     mov    eax,cr0
     or   al,1
     mov    cr0,eax

     db   66h
     db   0EAh
ENTRY_OFF dd   PROTECTED_MODE_ENTRY_POINT
     dw   CODE_SELEKTOR

align 8
GDT:
```

```
   NULL_descr  db   8 dup(0)
   CODE_descr  db   0FFh,0FFh,00h,00h,00h,10011010b,11001111b,00h
   DATA_descr  db   0FFh,0FFh,00h,00h,00h,10010010b,11001111b,00h
   VIDEO_descr db   0FFh,0FFh,00h,80h,0Bh,10010010b,01000000b,00h
   GDT_size equ    $-GDT

label GDTR fword
     dw   GDT_size-1
     dd   ?

use32

PROTECTED_MODE_ENTRY_POINT:
     mov    ax, DATA_SELEKTOR
     mov    ds, ax
     mov    es, ax
     mov    ss, ax
     mov    esp, STACK_BASE_ADDRESS

     call delta
        delta:
     pop ebx
     add ebx, USER_PM_CODE_START-delta

     mov    esi, ebx
     mov    edi, USER_PM_CODE_BASE_ADDRESS
     mov    ecx, USER_PM_CODE_SIZE
     rep    movsb

     mov    eax,USER_PM_CODE_BASE_ADDRESS
     jmp    eax

USER_PM_CODE_START:
ORG USER_PM_CODE_BASE_ADDRESS

     include 'PM_CODE.ASM';

USER_PM_CODE_END:
```

Теперь поясним некоторые моменты. Первое отличие, которое заметно в коде, это:

```
align 8
GDT:
   NULL_descr  db   8 dup(0)
```

Директива `align` выравнивает следующий за ней код по указанной границе, т. е. все смещение данных после этой директивы будет кратно указанному числу; в нашем примере таблица GDT будет выровнена по границе 4 байт – это сделано для увеличения скорости работы процессора. Место, пропущенное для выравнивания, будет заполнено ничего не делающими командами NOP.

После перехода в защищённый режим мы инициализируем стек. Его начало будет располагаться по адресу 200000h (т.е. 2 Мб). Для стека мы выделим 4 Мб: вниз 2 Мб и вверх 2 Мб. Основной код защищённого режима, который будет находиться в файле PM_CODE.ASM, в памяти будет по адресу 400000h. Поместим этот код туда:

```
call delta
delta:
pop ebx
add ebx, USER_PM_CODE_START-delta
```

Этот код на первый взгляд непонятен для непосвящённого. Командой CALL мы передаём управление следующей команде – но для чего это делается? Для того чтобы получить смещение метки `delta`; после этого мы вычисляем смещение кода, который надо поместить в память по адресу 400000h. Для того чтобы код, который находится в файле PM_CODE.ASM, не дал сбоя при работе, мы указываем директиву ORG для изменения значений меток. Метки, находящиеся после этой директивы, имеют такое значение, как будто они находятся после указанного директивой ORG адреса, но на размещение данных в файле эта метка не влияет. Примеры использования директивы ORG приведены в листинге 2.3.

Листинг 2.3. Использование директивы ORG

```
ORG 100h
db 10h dup (0)
metka1:     ; метка имеет адрес 110h
ORG 500h
metka2:     ; метка имеет адрес 500h
db 10h dup (0)
metka3:     ; метка имеет адрес 510h
```

После перемещения кода, который находится в файле PM_CODE.ASM, программа передаёт ему управление. В этом файле можно спокойно использовать метки, не преобразовывая их в линейные адреса по правилам режима реальных адресов.

Далее в листинге 2.4 приведён текст файла PM_CODE.ASM – он выводит на экран строку «123456789».

Листинг 2.4. Код, выполняющийся в защищённом режиме

```
    mov         ESI, message
    mov         EDI, 0B8000h
    mov         ECX,18

    rep         movsb
    jmp         $

message: db "152535455565758595"
```

Текст сообщения теперь задан как строка, в которой каждый второй символ – это цифра «5» (имеет значение 35h).

Эта программа будет каркасной. И во всех следующих программах мы будем писать код только в файле PM_CODE.ASM.

Программа написана, но как её запустить? После компиляции программы будет получен COM-файл, который используется в ОС MS-DOS. Операционная система Windows поддерживает запуск программ для DOS, но тем не менее, пока мы находимся в Windows, все попытки запустить программу тщетны: она завершится при попытке выполнить инструкцию LGDT, т. к. последняя является привилегированной. Программу надо запускать из режима реальных адресов, а именно из-под операционной системы MS-DOS. Есть два наиболее простых способа попасть в операционную систему MS-DOS: создать загрузочную DOS-дискету или загрузочную DOS-флешку. Вариант с загрузочным флеш-диском с операционной системой MS-DOS наиболее приемлемый, потому что дискеты сейчас в большей степени раритет, нежели реально используемый носитель данных. В обоих случаях дискету или флешку необходимо поставить в списке загрузочных устройств ранее жёсткого диска, на котором находится ваша основная операционная система.

Использовать настоящий компьютер для тестирования своих программ защищённого режима неудобно, т. к. перезагрузка компьютера – не очень быстрый процесс, да и программа не сразу будет работать так, как нужно. А любое неправильное действие программы будет приводить к зависанию компьютера либо к перезагрузке, и иногда может потребоваться несколько десятков перезагрузок компьютера до получения нужного результата. Для тестирования программ защищённого режима хорошо подходят эмуляторы виртуальных компьютеров. Использовать тяжёлые и сложные эмуляторы типа VMWare или VirtualBox нецелесообразно, во-первых, потому что они заточены под операционные системы Windows или Linux, а во-вторых, не очень удобны при тестировании программ, поскольку не обладают средствами отладки. Одним из наиболее удобных является бесплатный эмулятор с открытым исходным кодом QEMU: он имеет версии почти под все платформы и обладает высокой скоростью эмуляции.

Не менее удобен бесплатный эмулятор с открытым исходным кодом Bochs: он обладает средствами отладки. Эмуляция процессора у Bochs очень тщательная и позволяет эмулировать процессор, поддерживающий почти все современные технологии, даже если процессор вашего компьютера не обладает такой способностью (т. е. на Bochs можно запустить 64-битную операционную систему, даже если процессор на вашем компьютере этого не поддерживает). Однако данное преимущество компенсируется тем, что Bochs имеет самую низкую скорость эмуляции. Хорошо, что при тестировании нашим программ скорость выполнения – самое последнее, что нас будет волновать!

Эмуляторы VirtualBox, QEMU и Bochs являются полностью бесплатными и их можно найти на CD-диске, прилагающемся к этой книге (папка vm).

2.1.5. Резюме

В этом разделе мы изучили основы работы в защищённом режиме, а также способ формирования линейного адреса и алгоритм перехода в защищённый режим. В практической части нами была написана программа, переводящая процессор в

защищённый режим. В следующем разделе мы рассмотрим прерывания в защищённом режиме.

2.2. Прерывания в защищённом режиме

После получения базовых сведений о защищённом режиме мы приступаем к изучению важнейшего механизма любой операционной системы – обработки прерываний.

Прерывания – неотъемлемая часть любой операционной системы. Без прерываний операционная система не сможет работать с внешними устройствами и защищать свои данные от приложений.

2.2.1. Что такое прерывание?

Для того чтобы операционная система могла защитить свои данные от пользовательских приложений и принимать данные от внешних устройств, процессор генерирует прерывания при нарушениях защиты и при получении сигналов от контроллера прерываний, который принимает сигналы от внешних и системных устройств. При генерации прерывания процессором текущая программа прерывается (отсюда и название); при этом сохраняются адрес следующей инструкции и регистр флагов, а управление передаётся обработчику. После своей работы обработчик возвращает управление прерванной программе, и её выполнение возобновляется как ни в чём не бывало. Прерывания бывают трёх видов:

1. Аппаратные.
2. Программные.
3. Исключения.

Аппаратные прерывания генерируются контроллером прерываний; их количество зависит от самого контроллера прерываний. Программные прерывания генерируются командой INT n. Исключения генерируются самим процессором при попытке нарушить ограничения защиты или при возникновении ошибок во время выполнения программы.

В режиме реальных адресов может существовать 256 различных прерываний. В начале памяти по адресу 0000:0000 расположена таблица прерываний, состоящая из 256 4-байтовых адресов в виде `<dw сегмент>:<dw смещение>`. Каждый такой адрес указывает на процедуру, обрабатывающую прерывание. Такие процедуры называются обработчиками прерываний, а адреса в этой таблице – векторами. Как уже понятно, в режиме реальных адресов исключений не было, т. к. никакой защиты тоже не было. В защищённом режиме количество прерываний такое же, но таблица прерываний теперь может находиться по любому адресу и называется IDT (Interrupt Descriptor Table, таблица дескрипторов прерываний).

Каждое прерывание описывается не 4-байтовым адресом, а дескриптором шлюза. Что такое шлюз? *Шлюз* – это специальный системный объект, который обеспечивает защиту и позволяет программам на разных уровнях использовать прерывания. Обычно обработчик прерывания находится на нулевом уровне привилегий,

а вызывающий – выше, поэтому процессор как бы проходит через шлюз и опускается вниз на нулевой уровень привилегий, а после обработки поднимается вверх на уровень вызывающего.

Различают три типа шлюзов:

1. Шлюз задачи.
2. Шлюз прерывания.
3. Шлюз ловушки.

Отличия этих трёх типов будут приведены позже.

Бит IF регистра флагов отвечает за маскируемые прерывания. Если он очищен, то маскируемые прерывания игнорируются. Если он выставлен, то прерывания обрабатываются. *Маскируемые прерывания* называются так именно потому, что на них можно поставить маску и запретить некоторые. При сбросе флага IF запрещаются все маскируемые прерывания.

2.2.2. Дескрипторы шлюзов

Дескриптор шлюза сходен с дескриптором сегмента. Общий формат шлюзов приведён в табл. 2.1. Уровень привилегий дескрипторов определяет уровень привилегий программы, которая может воспользоваться этим прерыванием (т. е. дескриптором).

Таблица 2.1. Общий формат дескриптора шлюза

Биты	Номер двойного слова и биты в нём	Назначение
0–15	1:0–15	Зависит от типа
16–31	1:16–31	Селектор (зависит от типа)
32–36	2:0–4	Зарезервировано
37–39	2:5–7	Зависит от типа
40–42	2:8–10	Тип шлюза
43	2:11	Зависит от типа
44	2:12	Всегда 0
45, 46	2:13, 14	Уровень привилегий дескриптора
47	2:15	Бит присутствия
48–63	2:16–31	Зависит от типа

При использовании шлюза задачи в случае, если генерируется прерывание, которое обрабатывается шлюзом задачи, происходит автоматическое переключение задачи. В битах 0–15 содержится селектор TSS задачи, на которую будет производиться переключение при использовании этого прерывания. В поле «тип шлюза» будут стоять значения 1, 0, 1 в битах 8, 9, 10 соответственно. Все остальные биты за исключением бита присутствия всегда равны нулю либо зарезервированы. Более подробно этот тип будет рассматриваться в разделе 2.4. Формат дескриптора шлюза задачи приведён на рис. 2.5.

Рис. 2.5. Формат дескриптора шлюза задачи

Дескрипторы шлюза ловушки и дескрипторы шлюза прерывания почти полностью идентичны. Отличие состоит лишь в том, что в битах типа шлюза у дескриптора шлюза ловушки будет стоять значение 111, а у шлюза прерывания – 110. В поле 16–31 находится селектор сегмента кода, в котором находится обработчик прерывания. В полях 0–15 и 48–63 находятся младшие и старшие 16 бит адреса обработчика. Схемы работы прерываний немного отличаются друг от друга.

При переходе через шлюз прерывания процессор автоматически сбрасывает флаг IF и тем самым не допускает генерации других прерываний на время работы обработчика, а для шлюза ловушки – не меняет состояние флага IF. Ловушки используются для отладки программ, поэтому обработка ловушки должна быть прозрачна для внешних прерываний.

Бит D – это размер шлюза: если он выставлен, то 32 бита; иначе 16 бит. Размер шлюза определяет размер стека, используемый процессором по умолчанию. Перед вызовом обработчика процессор помещает в стек значения регистров CS, EIP, EFLAGS и SS, ESP (если переход осуществился на другой уровень привилегий) и 16-битный код ошибки. Если размер шлюза – 32 бита, то значения размером в 16 бит будут расширены нулями до 32. На рис. 2.6 и 2.7 приведены форматы дескриптора шлюза ловушки и дескриптора шлюза прерывания соответственно.

Если бит присутствия сброшен, то процессор понимает это как отсутствие обработчика и генерирует исключение общей защиты. Дескриптор обработчика исключения общей защиты находится в IDT на 13 месте (нумерация начинается с нуля). При любых непредвиденных ошибках и для большинства стандартных ошибок генерируется исключение общей защиты. Если генерируется конкретное исключение и его обработчика нет в IDT или в его дескрипторе ошибки, то генерируется исключение отсутствующего сегмента; если его дескриптора нет, то процессор «зависнет» или перезагрузится.

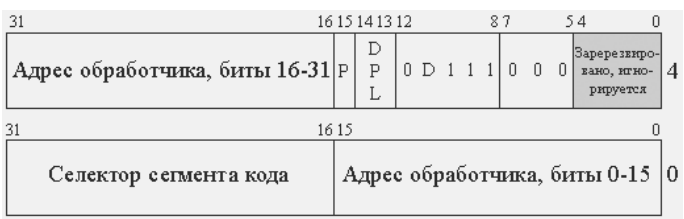


Рис 2.6. Формат дескриптора шлюза ловушки

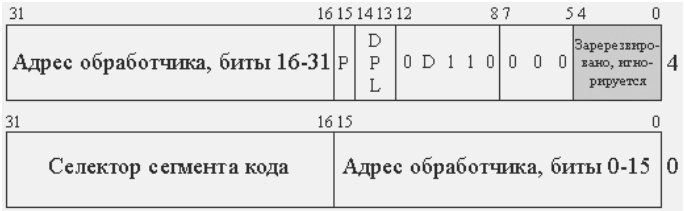


Рис. 2.7. Формат дескриптора шлюза прерывания

Поле DPL содержит максимальный уровень привилегий, с которого можно обратиться в этому шлюзу. Например, если поле DPL содержит число 1, то вызвать данный шлюз (команда `int <vector>`) могут только программы, выполняющиеся на уровнях привилегий 0 и 1; вызов этого шлюза на уровнях 2 и 3 приведёт к исключению общей защиты.

Все дескрипторы объединяются в таблицу IDT. В этой таблице может быть от 0 до 256 дескрипторов. Параметры IDT задаются в регистре IDTR аналогично регистру GDTR.

Рис. 2.8. Формат регистра IDTR

Команда LIDT загружает в регистр IDTR, SIDT сохраняет содержимое IDTR в память. Команда LIDT является привилегированной и может выполняться только на нулевом уровне привилегий; команда SIDT не является привилегированной. Как и в случае с командами для работы с GDT, для обеих этих команд операндом должны быть 6 байт в памяти.

2.2.3. Исключения

Исключения генерируются самим процессором, когда какая-либо программа пытается нарушать ограничения защиты. Повлиять на исключения прикладные программы (работающие даже на нулевом уровне привилегий) не могут, замаскировать – тоже. Исключения процессора генерируются вне зависимости от флага IF. Аппаратный контроль защиты – самый надёжный, и 32-разрядные процессоры предоставляют этот сервис в полном объёме.

Исключения делятся на три типа в зависимости от условий их возникновения:

1. *Ошибка* – это исключение, возникающее в ситуации ошибочных действий программы (подразумевается, что такую ошибку можно исправить). Такой тип исключения допускает рестарт команды, которая вызвала исключение после исправления ситуации, для чего в стеке обработчика адрес возврата из прерывания указывает на команду, вызвавшую исключение. Примером такого исключения может быть исключение отсутствующей страницы. Благодаря этому реализуется механизм виртуальной памяти, в частности подкачка данных с диска.
2. *Ловушка* – это исключение, возникающее сразу после выполнения «отлавливаемой» команды. Это исключение позволяет продолжить выполнение программы со следующей команды. На ловушках строится механизм отладки программ.

3. *Авария* – это исключение, которое не позволяет продолжить выполнение прерванной программы и сигнализирует о серьёзных нарушениях целостности системы. Примером аварии служит исключение двойного нарушения (прерывание 8), когда сама попытка обработки одного исключения вызывает другое исключение.

С целью корректного определения источника ошибки для некоторых исключений процессор помещает в стек 2-байтовый код ошибки. Ниже, в табл. 2.2, приведены исключения процессора. В графе «Название» приведены названия исключений в соответствии с документацией от Intel. Обычно это начальные буквы названия, например #PF (Page Fault – ошибка страницы). Исключений всего 32. Они проецированы на векторы 0–1Fh, и использовать эти векторы для других прерываний нельзя.

Таблица 2.2. Типы исключений в защищённом режиме

Номер вектора	Название	Описание	Тип	Код ошибки	Источник
0	#DE	Ошибка деления	Ошибка	Нет	Команды DIV и IDIV
1	#DB	Отладка	Ошибка или ловушка	Нет	Любая команда или команда INT 1
2	–	Прерывание NMI	Прерывание		Немаскируемое внешнее прерывание
3	#BP	Точка останова	Ловушка	Нет	Команда Int3
4	#OF	Переполнение	Ловушка	Нет	Команда INTO
5	#BR	Превышение предела	Ошибка	Нет	Команда BOUND
6	#UD	Недопустимая команда	Ошибка	Нет	Недопустимая команда или команда UD2
7	#NM	Устройство недоступно	Ошибка	Нет	Команды плавающей точки или команда WAIT/FWAIT
8	#DF	Двойная ошибка	Авария	Всегда ноль	Любая команда
9	–	Зарезервировано компанией Intel. Не использовать!			
10	#TS	Недопустимый TSS	Ошибка	Да	Переключение задач или доступ к TSS
11	#NP	Сегмент отсутствует	Ошибка	Да	Загрузка сегментных регистров или доступ к сегментам
12	#SS	Ошибка сегмента стека	Ошибка	Да	Операции над стеком и загрузка в SS
13	#GP	Общая защита	Ошибка	Да	Любой доступ к памяти и прочие проверки защиты

Таблица 2.2. Типы исключений в защищённом режиме

Номер вектора	Название	Описание	Тип	Код ошибки	Источник
14	#PF	Страничное нарушение	Ошибка	Да	Доступ к памяти
15	–	Зарезервировано компанией Intel. Не использовать!			
16	#MF	Ошибка плавающей точки в x87 FPU (ошибка математики)	Ошибка	Нет	Команда x87 FPU или команда WAIT/FWAIT
17	#AC	Проверка выравнивания	Ошибка	Всегда ноль	Обращение к памяти
18	#MC	Проверка оборудования	Авария	Нет	Наличие кодов и их содержимое зависит от модели
19	#XF	Исключение плавающей точки в SIMD	Ошибка	Нет	Команды SSE
20–31	–	Зарезервировано компанией Intel. Не использовать!			

Обработчики исключений не обязательно должны выполнять свои функции. На этапе создания операционной системы достаточно сделать так называемые «заглушки», которые будут выводить на экран номер исключения и параметры, переданные в стеке, и «подвешивать» процессор. Флаг IF не влияет на исключения. Следует подметить, что прерывание 2 является единственным немаскируемым прерыванием. За маскировку этого прерывания отвечает седьмой (нумерация начинается с нуля) бит порта 0A2h: если он выставлен, то прерывание NMI не генерируется.

Исключение двойной ошибки возникает в случае, когда при вызове обработчика какого-либо исключения опять возникло исключение. Третьим обработчиком будет вызван обработчик двойного исключения. Если при вызове обработчика двойного исключения опять возникло исключение, процессор выключается либо перезагружается.

2.2.4. Коды ошибок

Коды ошибок помещаются в стек при следующих исключениях (кроме тех, у которых код ошибки всегда ноль): недопустимый TSS (`#TS`), отсутствующий сегмент (`#NP`), ошибка стека (`#SS`), исключение общей защиты (`#GP`), страничное нарушение (`#PF`). Чаще всего в качестве кода ошибки передаётся селектор сегмента, в котором или из-за которого произошло исключение. Поле RPL в данном случае используется для других целей. Формат селектора ошибки в этих случаях приведён на рис. 2.9.

Бит EXT. Если он сброшен, то исключение было вызвано внутренними источниками; если он выставлен, то внешними.

Бит IDT. Если бит сброшен, селектор указывает на дескриптор, находящийся в GDT или IDT; если выставлен, то на дескриптор в IDT.

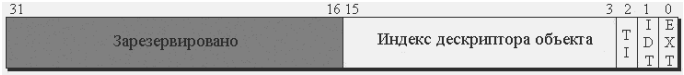


Рис. 2.9. Формат селектора в коде ошибки

Бит TI. Если бит сброшен, то селектор указывает на дескриптор, находящийся в GDT; если выставлен, то на дескриптор в LDT. Если бит IDT выставлен, то данный бит не имеет значения.

Формат кода ошибки страничного нарушения будет описан в разделе 2.3.

2.2.5. Программные прерывания

Прерывания можно генерировать программно. Команда INT генерирует прерывание. Число, указанное в качестве операнда, обозначает номер дескриптора в IDT, например: INT 25. Команда INT3 генерирует системное прерывание, которое вызывает прерывание 3. Команда INT3 имеет короткий опкод. Команда INTO вызывает прерывание 4, если установлен флаг OF.

Программные прерывания – это наиболее простой из методов вызова привилегированного кода для программ, которые работают на нулевом уровне привилегий. В системах до WinXP используется именно такой метод вызова привилегированного кода. На процессорах Pentium II и старше можно использовать более защищённый метод вызова привилегированного кода.

2.2.6. Аппаратные прерывания

Посредником между устройствами и процессором служит контроллер прерываний, который можно назвать «диспетчером». Он позволяет процессору обрабатывать прерывания согласно приоритетам и избежать возможных коллизий при поступлении сигналов сразу от нескольких источников. В компьютере есть программируемый контроллер прерываний (PIC), который тесно взаимосвязан с процессором. Этот контроллер довольно-таки старый, на современных системах используется расширенный программируемый контроллер прерываний (APIC).

Контроллер прерываний представляет собой два контроллера i8259A, соединённых вместе. Второй контроллер подключён ко второй линии первого контроллера. Такой метод подключения называется каскадным: один контроллер ведущий, другой – ведомый. Вместе они дают 16 линий прерываний. Такая ситуация была в 80-е годы. Сейчас на всех компьютерах установлены контроллеры APIC; они более гибкие и программируются совсем не так, как старые. Из соображений совместимости контроллером можно управлять как двумя i8259A, но воспользоваться всеми возможностями APIC не удастся.

В этом разделе будет изложен старый метод программирования контроллеров. Программирование APIC будет рассмотрено в главе 6. Описание работы с контроллером i8259A не входит в задачу этого раздела, поэтому здесь будут приведены только базовые сведения по работе с PIC.

Контроллер прерываний при инициативе устройства даёт запрос процессору об обработке прерывания. Если флаг IF выставлен, то процессор отвечает. При ответе процессора контроллер прерываний выставляет на шину данных номер

линии плюс некоторое число. Это число в режиме реальных адресов (вернее сказать, в операционной системе MS-DOS) равно 8 для ведущего контроллера, а для ведомого – 70h. В наших программах защищённого режима мы будем использовать 20h для ведущего контроллера и 28h – для ведомого. При поступлении нескольких заявок от разных источников они обрабатываются по порядку, начиная с меньшего номера IRQ. Можно также выборочно заблокировать некоторые заявки от отдельных IRQ-линий. За программирование контроллеров отвечают два регистра – iMr и iSr, которые подключены к двум портам ввода-вывода. Каждый контроллер имеет свои собственные регистры iMr и iSr.

Регистр iMr отвечает за маскировку прерываний. Каждый бит этого регистра отвечает за отдельную линию прерываний контроллера. Если бит выставлен, то эта линия запрещена. У каждого контроллера есть свой регистр iMr. У ведущего контроллера этот регистр подключён к порту 21h процессора, а у ведомого – к порту 0A1h процессора. Разумеется, если замаскировать вторую линию ведущего контроллера, то маскируются все линии ведомого.

Регистр iSr сообщает, какое сейчас обрабатывается прерывание. Если это прерывание 3, то третий бит выставлен. Сброс соответствующего бита должна осуществлять сама программа, т. е. обработчик. Установленный бит n регистра iSr блокирует прерывания от входов с номерами большими либо равными n. Также он оповещает контроллер о том, что прерывание обработано. Обычно для этой цели пользуются командой неопределённого сброса: вывод в порт iSr значения 20h. В конце обработки прерывания надо обязательно привести регистр iSr в соответствующее состояние, т. е. сказать контроллеру прерываний, что мы обработали прерывание. Вывод значения 20h в порт iSr – как раз то, что нам нужно для начала.

Регистр iSr ведущего контроллера подключён в порту 020h, а ведомого – к порту 0A0h. А регистры iMr контроллеров подключены к портам 021h и 0A1h соответственно. Для инициализации контроллеров используется код такого формата:

```
out iSr 00010001b
out iMr, <прибавляемое число при выдаче на шину данных>
; должно быть кратно 8
out iMr, 00000100b ; зависит от контроллера
out iMr, 00000001b
```

Далее приведён код инициализации ведущего контроллера. Разумеется, выводить в порт можно только из регистра EAX (AX, AL), но эти строчки опущены для лучшей читаемости:

```
out 20h, 00010001b
out 21h, 20h ; номер обработчика для нулевой линии
out 21h, 00000100b
out 21h, 00000001b
```

Третья строчка обозначает битовую маску подключения ведомого контроллера, а установленный бит маски – линию, к которому подключён ведомый контроллер. Далее приведён код инициализации ведомого контроллера.

```
out A0h, 00010001b
out A1h, 28h ; номер обработчика для нулевой линии
```

```
out A1h, 00000010b
out A1h, 00000001b
```

Третья строчка обозначает номер линии, через которую ведомый контроллер подключён к ведущему. После этого все 16 линий прерываний направлены на вектора 20–2Fh.

Как уже было сказано, флаг IF отвечает за маскируемые внешние прерывания. Если флаг IF сброшен, то процессор не отвечает на запросы контроллера прерываний.

И напоследок следует отметить, что прерывание 2 (немаскируемое прерывание NMI) используется как оповещение о том, что произошел аппаратный сбой. Основная особенность прерывания в том, что процессор при получении сигнала на прерывание по входу NMI не опрашивает контроллер прерываний на предмет получения номера обработчика, а сразу вызывает обработчик 2. Это нужно, чтобы «избежать контактов» с неисправным оборудованием. Маскировка прерывания NMI осуществляется установкой седьмого бита в порте 70h.

```
in  al, 70h
or  al, 80h
out 70h, al
```

После этого даже при возникновении прерывания NMI доставляться процессору оно не будет.

2.2.7. Обработчик прерывания

При генерации прерывания или исключения происходит следующее:

1. Из регистра IDTR извлекается база таблицы дескрипторов прерываний (IDT).
2. В таблице IDT по номеру прерывания находится дескриптор шлюза прерывания. Если его бит P сброшен, генерируется исключение общей защиты.
3. Если текущий уровень привилегий отличается от уровня привилегий обработчика, происходит переключение стека и в стеке обработчика сохраняется указатель на стек прерванной задачи (SS и ESP).
4. В стек помещаются регистры EFLAGS, CS, EIP. Для некоторых исключений последним в стек помещается еще и код ошибки, который, кстати, должен «вытолкнуть» обработчик исключения перед выходом.
5. В регистре флагов обработчика очищается бит TF (флаг трассировки), для программного прерывания или исключения сбрасываются биты VM (флаг режима виртуального 8086), RF (о нём мы поговорим чуть позже) и NT (об этом флаге речь пойдет в разделе 2.4).
6. При вызове обработчика через шлюз прерывания очищается бит IF, блокируя дальнейшие маскируемые аппаратные прерывания. При вызове обработчика шлюза ловушки этого не делается.

После обработки прерывания обработчик должен вытолкнуть из стека код ошибки, если он там есть, и выполнить инструкцию IRETD, которая восстанавливает регистр флагов из стека. Если уровень привилегий прерванной задачи не равен уровню привилегий обработчика, происходит переключение стека на стек

прерванной задачи с использованием значений регистров SS и ESP, которые были сохранены в стеке обработчика при вызове. Проще говоря, при выполнении команды IRETD производятся действия, обратные вызову обработчика прерывания.

Обработка аппаратных прерываний несколько отличается от обработки обычных прерываний, т. к. надо ещё уведомить контроллеры о том, что обработка прерывания закончилась. Далее приведён каркас стандартного обработчика аппаратных прерываний при использовании контроллера i8259A, с обработкой обычных прерываний:

Листинг 2.5. Шаблон обработчика аппаратных прерываний

```
Handler:
   ... ...
   ... ...
   push ax
   mov  al, 20h
   out  020h, al    ; сброс контроллеров
   out  0a0h, al
   pop  ax
   iretd
```

Сброс контроллеров также называется операцией EOI (End Of Interrupt).

Флаг RF отвечает за рестарт инструкции, вызвавшей исключение. Если флаг выставлен, то текущая инструкция (т. е. предшествующая инструкции, на которую указывает EIP) выполнится снова. Помимо того что данный флаг повторно выполняет текущую инструкцию, он запрещает отладочное исключение (#DB).

Процессор очищает этот флаг после выполнения каждой инструкции, за исключением двух случаев:

1. После выполнения команды IRET, которая установила данный флаг.
2. После команды JMP, CALL или INT n, если целью являлся шлюз задачи.

При генерации исключений в стек обработчика помещается регистр флагов с флагом RF = 1, и при возвращении управления команда, вызвавшая исключение, выполнится ещё раз. Также при генерации прерываний в стек обработчика помещается регистр флагов с флагом RF=1, если прерванная инструкция являлась инструкцией работы со строками (MOVS, LODS, CMPS, SCAS, STOS и OUDS).

Для исключений типа «ловушка» не подразумевается рестарт инструкции, поэтому в стек обработчика помещается регистр флагов с флагом RF = 0. Для рестарта инструкции надо вручную установить флаг RF в стеке обработчика в той позиции, куда помещается регистр флагов (в стеке обработчика после кода ошибки, если он есть, следует EIP прерванной программы, потом CS, потом EFLAGS).

2.2.8. Практика

Теперь закрепим все полученные знания на практике. Мы напишем программу защищённого режима, которая реагирует на нажатия клавиатуры и на сигналы системного таймера. Будем писать программу на основе той, которую мы создали в разделе 2.1. Нам потребуется только файл PM_CODE.ASM.

Для начала напишем макрос для объявления дескриптора.

Листинг 2.6. Макрос для объявления дескриптора шлюза

```
macro DEFINE_GATE _address,_code_selektor, _type
{
dw _address and 0FFFFh ,_code_selektor,_type, _address shr 16
}
```

Этот макрос объявляет дескриптор; третьим параметром он принимает тип дескриптора. Для дескриптора шлюза это будет:

```
INT_GATE equ 1000111000000000b
```

Теперь вся таблица прерываний:

Листинг 2.7. Объявление IDT с помощью макроса DEFINE_GATE

```
   align 8
IDT:
   dq 0 ; 0
   DEFINE_GATE syscall_handler, CODE_SELEKTOR,INT_GATE
   dq 0 ; 2
   dq 0 ; 3
...
   dq 0 ; 12
   DEFINE_GATE exGP_handler, CODE_SELEKTOR,INT_GATE ; 13 #GP
   dq 0 ; 14
...
   dq 0 ; 31
   DEFINE_GATE irq0_handler, CODE_SELEKTOR,INT_GATE;20h IRQ0
   DEFINE_GATE irq1_handler, CODE_SELEKTOR,INT_GATE;21h IRQ1
   DEFINE_GATE int_EOI, CODE_SELEKTOR,INT_GATE ; 22h IRQ2
...
   DEFINE_GATE int_EOI, CODE_SELEKTOR,INT_GATE ; 2Fh IRQ15
   IDT_END:
```

Размер IDT вычисляется следующим образом:

```
IDT_size  equ IDT_END-IDT
```

Собственно, загрузка IDTR будет выглядеть так:

```
lidt       fword [IDTR]
```

Идём далее. Теперь нам надо перенаправить прерывания. Для того чтобы это сделать, нам надо заставить контроллер прерываний изменить число, которое прибавляет к номеру прерывания контроллер прерываний перед подачей на шину данных, т. е. перенаправить прерывания на векторы 20–2Fh. Начальные векторы для ведущего и ведомого контроллера мы поместим в регистры в BL и BH соответственно (листинг 2.8).

Листинг 2.8. Перенаправление векторов контроллера прерываний

```
mov  bx, 2820h

mov  al, 00010001b
```

```
out  020h, al
out  0A0h, al
mov  al, bl
out  021h, al
mov  al, bh
out  0A1h, al
mov  al, 00000100b
out  021h, al
mov  al, 2
out  0A1h, al
mov  al, 00000001b
out  021h, al
out  0A1h, al
```

После этого нам надо разрешить все прерывания. Аппаратные прерывания разрешаются установкой флага IF, а прерывание NMI – сбросом старшего бита в порте 70h (листинг 2.9).

Листинг 2.9. Разрешение аппаратных прерываний

```
in   al, 70h
and  al, 7Fh
out  70h, al
sti
```

Теперь напишем пустой обработчик, который поставим в качестве обработчика всех прерываний, кроме двух первых (первые 2: таймер и клавиатура) – листинг 2.10.

Листинг 2.10. Пустой обработчик аппаратных прерываний

```
int_EOI:
   push ax
   mov  al, 20h
   out  020h, al
   out  0a0h, al
   pop  ax
   iretd
```

Этот обработчик отправляет обоим контроллерам сигнал о том, что прерывание обработано. Далее в листинге 2.11 приведён обработчик прерываний таймера.

Листинг 2.11. Обработчик прерываний таймера

```
irq0_handler:
   push eax
   push edx
   push ebx

   xor edx, edx
   inc dword [counter]
   mov eax, dword [counter]
   mov ebx, 18;
```

```
  div ebx
  cmp edx, 0
  jnz .cont

  inc  byte [es:6]

  cmp  byte [es:6],":"
  jnz .cont
  mov byte [es:6],"0"
  inc byte [es:4]

  cmp byte [es:4], ":"
  jnz .cont
  mov byte [es:4],"0"
  inc byte [es:2]

  cmp byte [es:2], ":"
  jnz .cont
  mov byte [es:2],"0"
  inc byte [es:0]

  cmp byte [es:0], ":"
  jnz .cont
  mov byte [es:0], "0"
  mov byte [es:2], "0"
  mov byte [es:4], "0"
  mov byte [es:6], "0"

.cont:
  pop ebx
  pop edx
  pop eax

  jmp int_EOI
```

В обработчике таймера мы увеличиваем счётчик секунд. По умолчанию он генерирует сигнал 18 раз в секунду. На экране в верхнем левом углу у нас будет число секунд, которое прошло после перехода в защищённый режим (максимум 9999 секунд, примерно 2,7 часа). При каждом срабатывании прерывания мы увеличиваем значение счётчика; если оно кратно 18, то мы увеличиваем значение счётчика на экране. Увеличение высшего разряда достигается за счёт сравнивания значения текущего разряда со значением : (двоеточие); т. к. по коду ASCII этот символ следует сразу за символом 9, после увеличения старшего разряда на 1 мы сбрасываем текущий. После окончания обработки прерывания мы передаём управление пустому обработчику, который в свою очередь передаёт сигнал обоим контроллерам о том, что прерывание обработано (листинг 2.10).

Код обработчика клавиатуры здесь приводиться не будет, поскольку написание драйвера клавиатуры не входит в задачу данного раздела (см. полный исходник примера). В листинге 2.12 приведён код программы после разрешения прерываний.

Листинг 2.12. Код программы после разрешения прерываний

```
mov eax, 00F00h
mov ecx, 0820h
mov edi, 00
rep stosw

mov byte [es:0], "0"
mov byte [es:2], "0"
mov byte [es:4], "0"
mov byte [es:6], "0"

mov  dword [cursor], 80

mov esi, message1
int 1
mov  dword [cursor], 160
mov esi, message2
int 1
mov  dword [cursor], 240

jmp  $
```

Сначала очищается экран, так чтобы были белые буквы на черном фоне, заполняя видеопамять значениями 0F00h. Мы знаем, что значения в памяти хранятся в обратном порядке; следовательно, 0Fh интерпретируется как цвет символов (чёрный фон, белый символ). Потом пишется в углу экрана текущее значение секунд. Далее с помощью системного вызова `int 1` мы выводим две строки. Теперь осталось только написать системный вызов `int 1`, обеспечивающий вывод строки, адрес которой передаётся через регистр ESI (листинг 2.13).

Листинг 2.13. Код обработчика прерывания на векторе 1

```
syscall_handler:
    pushad
  _puts:
    lodsb
    mov  edi, dword [cursor]
    mov  [es:edi*2], al
    inc  dword [cursor]
    test al, al
    jnz  _puts
    popad
    iretd
```

Текст листинга 2.13 тривиален – он выводит строку в область памяти, на которую указывает сегментный регистр ES (текстовый видеобуфер). Обработка специальных символов (символов перевода каретки и новой строки) не производится; впрочем, она и не нужна. Переменная cursor содержит текущее положение курсора на экране. Конец строки определяется по нулевому символу.

Расположение обработчика на векторе, зарезервированном компанией Intel для прерываний отладки, – не очень правильный шаг, но в данном примере прерывание отладки всё равно не используется, и это более удобно, нежели ставить обработчик, выводящий текст на вектор после векторов аппаратных прерываний.

Полный код программы находится в папке part2 на компакт-диске, прилагающемся к данной книге.

2.2.9. Резюме

В этом разделе мы изучили обработку прерываний и исключений в защищённом режиме и для закрепления изученного материала написали программу, работающую с прерываниями. В разделе 2.3 мы будем говорить о втором по важности механизме любой операционной системы – о механизме трансляции виртуальных адресов в физические.

2.3. Механизм трансляции адресов

Одно из самых главных нововведений защищённого режима – механизм трансляции виртуальных адресов в физические (механизм трансляции страниц). Благодаря механизму трансляции страниц операционная система может защитить свои данные от пользовательских программ и создать у вас ощущение, что для каждой программы отводится обособленное адресное пространство, а также что на компьютере наличествуют все 4 Гб памяти – даже если в действительности это не так.

Механизм трансляции страниц позволяет преобразовать виртуальные адреса в физические, при этом виртуальный адрес может и не совпадать с физическим. Программы при этом даже «не подозревают» о том, что они реально обращаются к другим адресам: адреса преобразовывает сам процессор аппаратным образом.

При включении специального режима расширенной физической адресации (PAE) процессор может адресовать вплоть до 64 Гб физической памяти. Но не будем забегать вперёд – начнём по порядку с обычного режима трансляции страниц.

2.3.1. Что это такое?

Сделаем небольшое лирическое отступление. При обращении к памяти процессор всегда использует пару значений – селектор и смещение (селектор однозначно определяет сегмент памяти). Как правило, селектор находится в сегментном регистре, смещение – в регистре общего назначения, а при обращении к памяти программисты стараются использовать селекторы по умолчанию, т. к. такие команды выполняются максимально быстро. Также адрес в памяти можно указать другими способами – через константы и значения переменных, но независимо от того, какой способ адресации мы применили при обращении к памяти, всегда используется пара значений «селектор и смещение» в явном или неявном виде. Адрес, указанный таким образом, называется *логическим адресом*, т. к. смещение задаётся относительно начала сегмента, который в свою очередь может быть размещён по любому адресу.

Каждый раз при выполнении команды процессор производит преобразование логического адреса в линейный адрес – 32-разрядный абсолютный адрес в памяти.

С линейными адресами вы уже сталкивались, когда определяли дескрипторы и регистры IDTR и GDTR, задавая базовые адреса этих объектов.

После вычисления линейного адреса процессор преобразует его в физический адрес, по которому и производит обращение к памяти.

В примерах из разделов 2.1 и 2.2 мы не использовали механизм трансляции адресов, поэтому все линейные адреса у нас были равны физическим. При включенном механизме трансляции адресов линейный адрес можно называть виртуальным, т. к. именно он преобразовывается в физический. Виртуального адреса может и не быть в физической памяти – создаётся лишь имитация его существования, отсюда и его название.

Теперь ближе к делу. Минимальной единицей измерения памяти является страница. *Страницей* называется непрерывная область памяти фиксированного размера. Впервые страничная организация памяти появилась в процессоре Intel386, где представилась возможность использования страниц размером 4 Кб. С появлением Pentium и Pentium Pro стало возможным использовать страницы размером 4 Мб (при обычном режиме доступны размеры страниц 4 Кб и 4 Мб, а при режиме PAE – 4 Кб и 2 Мб).

Итак, страница – это 4-килобайтная область памяти. Всё адресное пространство можно разбить на страницы (что, собственно говоря, и делается), так что в конечном счете получается 2^{20} страниц (т. е. 1048576), которые полностью покрывают 32-разрядное адресное пространство. Адрес страницы памяти всегда кратен 4 Кб (1000h байт), т. е. страница может начинаться только с адреса, у которого 12 младших бит равны нулю.

У каждой страницы есть два важных свойства: виртуальный и физический адреса. Если взять несколько страниц, то у каждой из них будут разные виртуальные адреса, но физические могут совпадать. Например, программа обращается к адресу 8D138Ch; тогда виртуальный адрес страницы, в которой находится данный адрес, будет 8D1000h. Физический адрес может отличаться от виртуального, но он также должен быть кратен 1000h, допустим 7F4000. В результате физический адрес будет равен 7F438Ch.

Каждую страницу можно пометить как отсутствующую в памяти. При недостатке физической памяти операционная система может сбросить содержимое некоторых страниц на внешний носитель (например, HDD), пометив при этом сброшенные страницы как отсутствующие; после этого освобождается память для других страниц. Обращение к отсутствующей странице приводит к генерации исключения ошибки страницы – #PF; после этого обработчик исключения возвращает содержимое страницы из внешнего носителя в свободную память (при необходимости выгрузив другую) и производит рестарт инструкции, вызвавшей исключение. При этом программа даже ничего «не подозревает» о том, что произошло.

За включение механизма трансляции адресов отвечает самый старший бит регистра CR0 под названием PG. Если он сброшен, то все линейные адреса являются физическими; если выставлен, то включается механизм трансляции адресов. Для включения специального режима расширенной физической адресации (PAE), с

помощью которой можно адресовать до 64 Гб физической памяти, используется бит PAE (5 бит регистра CR4). Сначала рассмотрим обычный режим.

2.3.2. Обычный режим трансляции адресов

2.3.2.1. Страницы размером 4 Кб

Каждая страница описывается структурой размером 4 байта. Все эти структуры объединяются в специальную таблицу под соответствующим названием – *таблица страниц*. Таблица страниц имеет размер 4 Кб, т. е. в ней можно описать 1024 страницы. Следовательно, с помощью одной таблицы страниц можно описать 2^{22} байт памяти, т. е. 4 Мб памяти. Для указания конкретного элемента таблицы страниц нам нужно значение размером в 10 бит ($2^{10} = 1024$).

Для описания всех 4 Гб памяти нам понадобятся 1024 таблицы страниц. Каждая таблица страниц описывается структурой размером 4 байта; все эти структуры объединяются в *каталог страниц*. Для указания конкретного элемента каталога страниц нам тоже нужно значение размером 10 бит.

При каждом обращении к памяти виртуальный адрес делится на три части, размеры которых – 10–10–12 (слева направо). Первая часть интерпретируется как индекс элемента в каталоге страниц. Из этого элемента извлекается физический адрес таблицы страниц. Вторая часть интерпретируется как индекс в этой таблице страниц. Из этого элемента извлекается физический адрес страницы. Третья часть интерпретируется как смещение в этой странице. На рис. 2.10 приведена схема трансляции виртуального адреса в физический.

Теперь надо описать структуры, участвующие в преобразовании адреса. Структура одного элемента таблицы страниц приведена на рис. 2.11.

Биты 12–31 – это начальный физический адрес страницы; разумеется, к нему добавляются справа 12 бит, заполненных нулями, т. к. каждая таблица страниц должна быть выровнена по границе страницы. На рис. 2.12 приведена структура одного элемента каталога страниц:

Рис. 2.10. Трансляция виртуального адреса в физический в обычном режиме

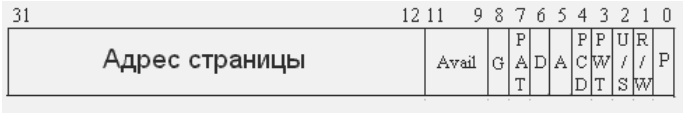


Рис. 2.11. Формат элемента таблицы страниц

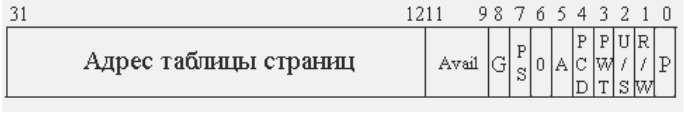


Рис. 2.12. Структура одного элемента каталога страниц

Биты 12–31 – это начальный *физический* адрес таблицы страниц. Далее приводятся описания полей.

Биты 9–11 обеих структур (элемента каталога страниц и элемента таблицы страниц) не используются процессором и могут быть использованы операционной системой в её целях.

Бит P. Если 0, то страница не отображена на физическую память. Это значит, что либо она не определена, либо её содержимое было записано на внешний носитель операционной системой. Если происходит обращение к отсутствующей странице (у которой бит P = 0), то процессор генерирует исключение ошибки страницы. Для каталога страниц это означает, что таблица страниц отсутствует в памяти, т. к. она помещается на одной странице; поэтому данный бит имеет то же значение, что и в элементе таблицы страниц. Если этот бит сброшен, то все остальные биты и поля игнорируются процессором.

Бит R/W. Этот бит показывает, доступна ли данная страница для записи. Если бит выставлен, то данные на этой странице можно изменить, иначе возможно только чтение. Для каталога страниц в случае, если этот бит сброшен, все страницы, описанные таблицей страниц, которую описывает данный элемент, предназначены только для чтения. Страница может быть доступной для записи, только если этот бит выставлен и в элементе каталога страниц, и в элементе таблицы страниц.

Бит U/S. Этот бит разрешает пользоваться страницей обычным приложениям. Если этот бит выставлен, то этой страницей (или группой страниц, если мы имеем дело с каталогом страниц) могут пользоваться приложения всех уровней. Если этот флаг сброшен, то страницей не могут пользоваться приложения третьего уровня привилегий. Пользовательский код может воспользоваться страницей, только если этот бит выставлен в элементах каталога страниц и таблицы страниц одновременно. Пользовательским кодом является код, который находится на третьем уровне привилегий.

Биты PWT и PCD. Эти биты управляют кэшированием страниц. Если бит PCD выставлен, то кэширование страницы (или группы страниц, если мы имеем дело с каталогом страниц) будет запрещено. Бит PWT включает/отключает сквозную запись (т. е. запись одновременно и в кэш и в память) для данной страницы (группы страниц). В этом разделе управление кэшированием рассматриваться не будет.

Бит A. Этот бит показывает, был ли произведён какой-либо доступ к данной странице со времени последнего сброса этого бита. Если он выставлен, то доступ был. Процессор этот бит только устанавливает, а сбрасывать его должна сама операционная система. Например, если операционная система хочет выгрузить страницу на внешний носитель, она может «посмотреть» этот бит и, если он не выставлен, не выгружать страницу на внешний носитель, поскольку в её отношении не было произведено ни одной операции чтения/записи (разумеется, при загрузке страницы с внешнего носителя этот бит должен сбрасываться). Если мы имеем дело с каталогом страниц, этот бит устанавливается при доступе к какой-либо странице таблицы, которая описывается данным элементом.

Бит D. Аналогичен предыдущему биту, но устанавливается всякий раз, когда происходит запись на эту страницу.

Бит PS. Этот бит задаёт размер страницы. Если он выставлен, то элемент каталога страниц указывает на страницу размером 4 Мб. Для страниц размером 4 Кб этот бит в элементе каталога страниц должен быть сброшен.

Биты PAT и G. Для процессоров, которые используют таблицу атрибутов страниц, флаг PAT используется совместно с флагами PCD и PWT для выбора элемента в PAT, который выбирает тип памяти для страницы. Этот бит введён в процессоре Pentium III, а для процессоров, не использующих PAT, бит PAT должен быть равен 0. Когда бит G выставлен, страница является глобальной. Такая страница остаётся действительной в кэшах TLB при перезагрузке регистра CR3 или переключении задач. В элементе каталога страниц этот бит игнорируется. Поскольку в данном разделе кэширование не обсуждается, подробно эти флаги рассматриваться не будут.

2.3.2.2. Страницы размером 4 Мб

Для страниц размером 4 Мб не нужна таблица страниц – достаточно всего лишь каталога страниц. Виртуальный адрес делится на две части 10–22. Первая часть интерпретируется как индекс элемента в каталоге страниц. После этого извлекается физический адрес из элемента, к этому адресу прибавляются младшие 22 бита виртуального адреса и в результате получается физический адрес. Ниже, на рис. 2.13, приведена схема трансляции адреса при использовании страниц размером 4 Мб.

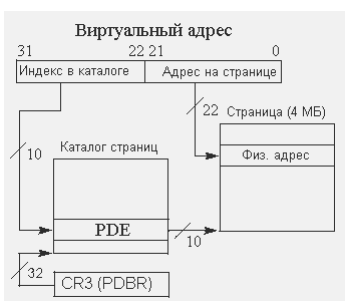


Рис. 2.13. Трансляция линейного адреса в физический при использовании страниц размером 4 Мб

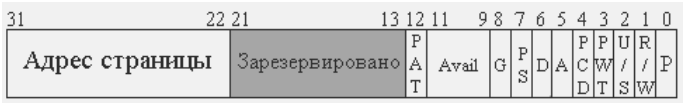


Рис. 2.14. Формат элемента каталога страниц при использовании страницы размером 4 Мб

Фактически каталог страниц выступает в роли таблицы страниц, т. к. каждый элемент в нём сразу указывает на страницу подобно элементу в таблице страниц.

На рис. 2.14 приведена структура элемента каталога страниц.

Как видно, младшие 12 бит адреса имеют тот же формат, что и формат каталога страниц при размере страниц 4 Кб.

Для включения возможности использования страниц размером 4 Мб надо выставить флаг PSE в регистре CR4. Бит PSE – это четвертый бит регистра CR4. Когда он выставлен, разрешается использовать страницы размером 4 Мб.

При трансляции адреса процессор «смотрит» значение бита PS элемента каталога страниц, и, если этот бит выставлен, элемент интерпретируется как указатель на страницу размером 4 Мб. Если бит сброшен, то элемент интерпретируется как указатель на таблицу страниц.

2.3.2.3. Регистр CR3

Регистр CR3 также называется PDBR (Page Directory Base Register). Для преобразования виртуального адреса процессору нужен каталог страниц – получая индекс каталога страниц, процессор получает адрес таблицы страниц, а по индексу в таблице страниц получает адрес страницы. Таким образом, процессору надо «знать» всего лишь адрес каталога страниц, а дальше он разберётся сам. В регистре CR3 содержится физический адрес каталога страниц, выровненный на границу страницы, т. е. все его 12 младших бит адреса не имеют значения. Формат регистра CR3 приведён на рис. 2.15.

Биты PWT и PCD отвечают за сквозную запись и контроль кэширования соответственно. В этом разделе кэширование мы рассматривать не будем, поэтому пока все младшие 12 бит регистра CR3 будут сброшены. С регистром CR3, как и с другими регистрами управления, можно работать только с помощью команды MOV; вторым операндом может быть только регистр общего назначения. Пример загрузки в CR3 некоторого значения:

```
mov eax, PDE_BASE
mov cr3, eax
```

Получение адреса текущего каталога страниц:

```
mov eax, cr3
```

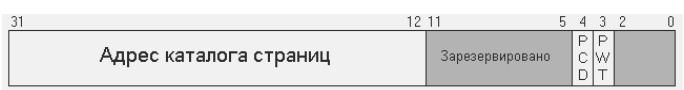


Рис. 2.15. Структура регистра CR3

Следует помнить, что после загрузки значения в регистр CR3 значения адреса надо снова выставить нужные флаги, если они были до этого установлены, либо загружать адрес каталога плюс 8, 16 или 24.

2.3.3. Режим расширенной физической трансляции адресов

Режим расширенной физической трансляции адресов также называется режимом PAE (Physical Address Extension). В этом режиме трансляции адресов операционная система может воспользоваться до 64 Гб физической памяти, при включении режима PAE физические адреса расширяются до 36 бит. Но для каждого приложения всё равно остаётся доступным только 4 Гб виртуальной памяти. При этом можно одновременно держать в физической памяти всю память нескольких приложений, которые используют максимум памяти, не выгружая их на внешний носитель.

Режим PAE включается установкой пятого бита в системном регистре CR4.

```
mov eax, cr4
or eax, 0100000b
mov cr4, eax
```

В режиме PAE можно использовать только страницы размером 4 Кб и 2 Мб. Когда включён режим PAE (т. е. пятый бит в регистре CR4 выставлен), то бит PSE (четвертый бит в регистре CR4) игнорируется процессором. В режиме PAE можно включить дополнительную защиту страниц (если, конечно, это поддерживается процессором), представляющую собой запрет на выполнение кода на конкретной странице.

2.3.3.1. Страницы размером 4 Кб

Для описания страниц в режиме PAE также используется таблица страниц, но теперь каждая страница описывается структурой размером 8 байт. Таблица страниц по прежнему имеет максимальный размер 4 Кб; следовательно, количество элементов в таблице страниц будет 512. Следовательно, память, которую можно адресовать с помощью одной таблицы, равна 2^{21} байт. Таблицы страниц объединяются в каталог страниц. Каждая таблица страниц тоже описывается элементом размером 8 байт; следовательно, и количество элементов тоже 512. С помощью одного каталога страниц можно адресовать 2^{30} байт памяти (1 Гб). Для описания всех 4 Гб виртуальной памяти нам потребуется четыре каталога страниц. Для определения каталога страниц тоже используется структура размером 8 байт. Четыре таких структуры объединяются в таблицу указателей на каталоги страниц.

При каждом обращении к памяти виртуальный адрес делится на четыре части размерами 2–9–9–12 бит слева направо. Первое 2-битовое поле интерпретируется как индекс элемента в таблице указателей на каталоги страниц. Из этого элемента извлекается физический адрес каталога страниц. Следующие 9 бит интерпретируются как индекс элемента в каталоге страниц. Из элемента каталога страниц извлекается физический адрес таблицы страниц. Следующие 9 бит интерпретируются как индекс в таблице страниц. Наконец, из элемента таблицы страниц извлекается физический адрес страницы и к нему прибавляется значение из 12 младших бит

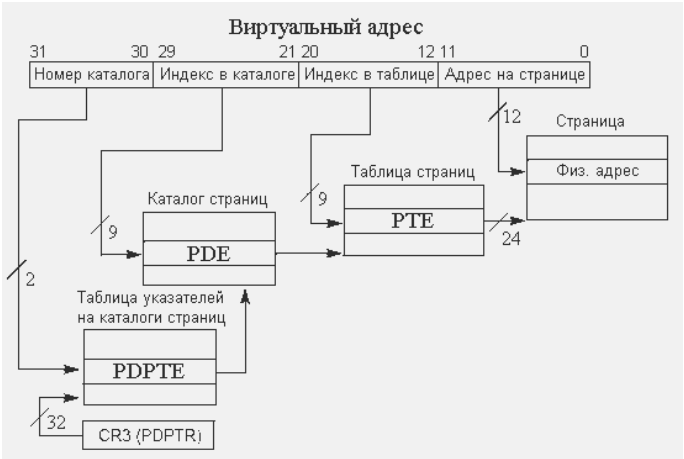


Рис. 2.16. Трансляция адреса в режиме PAE при использовании страниц размером 4 Кб

виртуального адреса. Ниже приведена схема преобразования виртуального адреса в физический в режиме расширенной физической адресации (рис. 2.16).

Как видно по рисунку, количество страниц такое же, как и в обычном режиме трансляции адресов.

Структуры, используемые при трансляции адреса, расширены до 8 байт для возможности хранения 36-битного адреса. Формат элемента таблицы указателей на каталоги страниц приведён на рис. 2.17.

Формат элемента каталога страниц представлен на рис. 2.18.

Следует подметить, что именно седьмой бит означает, что элемент указывает на таблицу страниц, а не на страницу размером 2 Мб. Элемент таблицы страниц приведён на рис. 2.19.

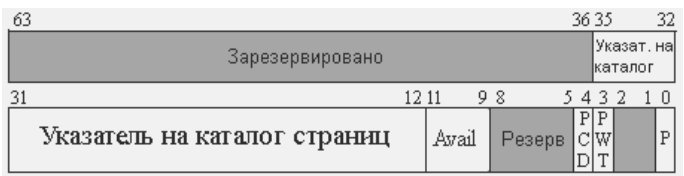


Рис. 2.17. Формат элемента таблицы указателей на каталоги страниц

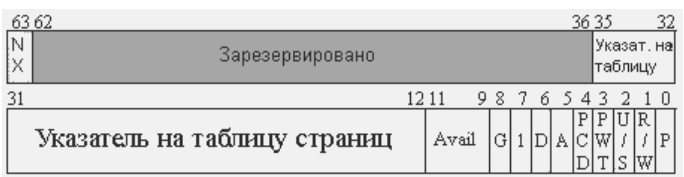


Рис. 2.18. Формат элемента каталога страниц в режиме PAE при использовании страниц размером 4 Кб

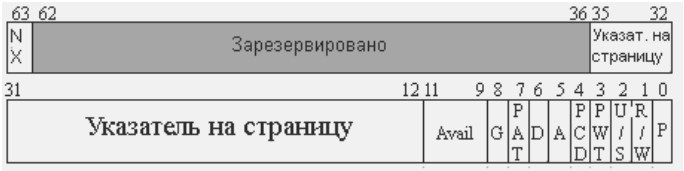


Рис. 2.19. Формат элемента таблицы страниц в режиме PAE

Назначения полей в этих структурах полностью идентичны назначению одноимённых полей в структурах, используемых в обычном режиме трансляции страниц. Новый элемент здесь – это бит NX, подробнее о нём будет рассказано в параграфе 2.3.3.4.

2.3.3.2. Страницы размером 2 Мб

Для возможности использования страниц такого размера не надо активировать никаких дополнительных режимов. Формат элемента таблицы указателей на каталоги страниц остаётся тем же, но формат элемента каталога страниц изменится (рис. 2.20).

Следует заметить, что, если седьмой бит сброшен, то элемент интерпретируется как указатель на таблицу страниц.

Если седьмой бит выставлен, то процессор интерпретирует младшие 21 бит виртуального адреса как смещение на странице, на которую указывает элемент каталога страниц. Схема преобразования виртуального адреса для страниц размером 2 Мб приведена на рис. 2.21.

2.3.3.3. Регистр CR3

В режиме PAE регистр CR3 содержит 32-битный адрес таблицы указателей на каталоги страниц, выровненный на границу 32 байт; следовательно, младшие 5 бит адреса ничего не будут значить. В режиме PAE нет необходимости размещать таблицу указателей на каталоги страниц по адресу кратному 4 Кб; достаточно разместить её по адресу кратному 32 байтам. Но при этом нет возможности разместить её по адресу выше 4 Гб.

2.3.3.4. Бит NX

Если этот бит NX выставлен, то на странице, описываемой данной структурой, запрещено выполняться любому коду; также запрещено передавать управление коду, который находится на этой странице. Для того чтобы какой-либо код мог выполняться на этой странице, данный бит должен быть сброшен как в каталоге

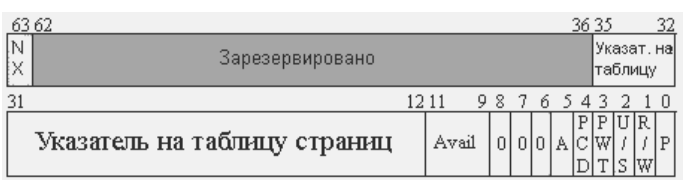


Рис. 2.20. Формат элемента каталога станиц в режиме PAE при использовании страниц размером 2 Мб

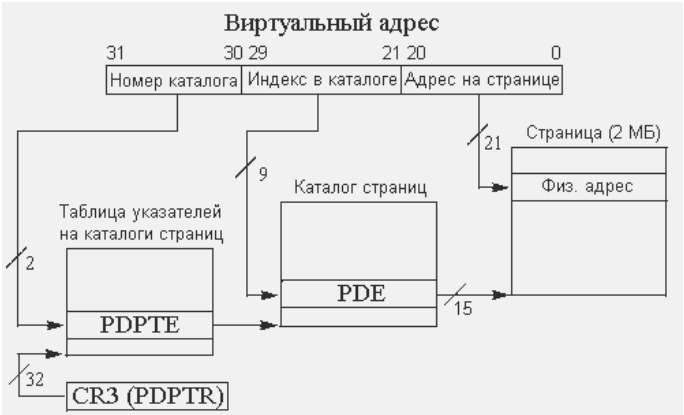


Рис. 2.21. Схема трансляции адреса в режиме PAE при использовании страниц размером 2 Мб

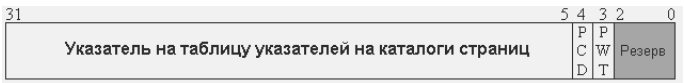


Рис. 2.22. Структура регистра CR3 в режиме PAE

страниц, так и в таблице страниц. Этот бит был введён как дополнительная защита от вредоносного кода.

Проверка поддержки бита NX осуществляется через инструкцию CPUID. Для этого надо вызвать эту инструкцию, указав в регистре EAX значение 80000001h. Если двадцатый бит в регистре EDX выставлен, то процессор поддерживает данную возможность. Но если просто выставить бит NX в структурах, этого будет недостаточно. Для того чтобы эта возможность по настоящему заработала, следует установить в единицу бит NXE (одиннадцатый бит) в MSR-регистре IA32_EFER. Если процессор не поддерживает бит NX, то попытка использовать каталоги или таблицы страниц с выставленным битом NX приведёт к генерации исключения общей защиты.

2.3.4. Обработчик страничного нарушения

Обработчик страничного нарушения расположен на 14 векторе в IDT. Исключение имеет класс «ошибка», т. е. после обработки исключения (если исключение не угрожает целостности данных) можно произвести рестарт команды, которая вызвала исключение. В стек обработчика помещается код ошибки, а в регистр CR2 – виртуальный адрес, из-за которого было вызвано страничное нарушение. Ниже приведён формат кода ошибки страничного нарушения (рис. 2.23).

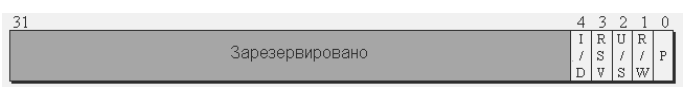


Рис. 2.23. Формат кода ошибка для обработчика страничного нарушения

Бит P. Если этот бит сброшен, то страничное нарушение вызвано отсутствующей страницей, а если выставлен – то нарушением доступа.

Бит R/W. Если этот бит выставлен, то ошибка произошла при записи, а если сброшен – то при чтении.

Бит U/S. Если этот бит выставлен, то ошибка была сгенерирована пользовательским кодом, а если сброшен – то кодом с уровнем привилегий 0,1 или 2.

Бит RSV. Если этот бит выставлен, то ошибка возникла после того, как процессор обнаружил, что зарезервированные биты в каталоге и таблице страниц имеют значение 1. Этот тип ошибки имеет место, только если выставлен какой-либо из флагов PSE или PAE.

Бит I/D. Имеет место только при включенной защите от выполнения кода на страницах, где это запрещено. Если бит выставлен, то ошибка вызвана при передаче управления на страницу, на которой запрещено выполнение кода.

В большинстве случаев главной функцией обработчика страничного нарушения является создание нового элемента в таблице страниц (если это необходимо), поиск незанятой физической страницы и проецирование новой страницы на найденную физическую. Также задачей обработчика страничного нарушения часто является загрузка содержимого страницы с внешнего носителя в память. Если нарушение произошло из-за нарушения доступа, то обработчик может просто аварийно завершить программу, но это уже зависит от самой операционной системы. После выполнения обработчика подразумевается рестарт инструкции, т. к. в стеке обработчика сохраняется адрес инструкции, следующей сразу за инструкцией, которая вызвала исключение.

Основная проблема обработчика страничного нарушения заключается в том, чтобы узнать, занята ли физическая страница. Одно из возможных решений – создание битовой карты. *Битовая карта* – это область данных, в которой каждый бит обозначает занятость страницы. Если, допустим, пятый бит карты выставлен, то страница занята. Для описания 1 Мб памяти нам потребуется 32 байта таблицы (1048576/4096/8 = 32). Если на компьютере установлено 512 Мб памяти, нам потребуется 16 Кб битовой карты, для описания 4 Гб памяти – 128 Кб битовой карты, а для описания 64 Гб памяти – 2 Мб битовой карты. Единственный минус этого метода – большой размер карты, и «прочёсывание» 128 Кб – не такое быстрое занятие, но можно придумать много алгоритмов для поиска нулевого бита в карте.

Для ускорения производительности системы рекомендуется создавать код обработчиков страничного нарушения наиболее быстрым и компактным. Разумеется, наилучшим вариантом будет написание обработчика страничного нарушения полностью на ассемблере. Также рекомендуется в начале кода обработчика получать и сохранять адрес, из-за которого произошло исключение, в регистр общего назначения либо в какую-либо переменную, т. к. во время работы обработчика может сгенерироваться ещё одно исключение страничного нарушения, и после этого изначальное значение адреса в CR2 будет потеряно.

2.3.5. Флаг WP в регистре CR0

По умолчанию привилегированный код (т. е. код, находящийся на уровнях 0, 1, 2) имеет право производить запись даже на те страницы, которые помечены только

для чтения. Но имеется возможность включить для привилегированного кода защиту от записи на страницы, помеченные только для чтения. Для этого надо выставить бит WP (16 бит) в регистре CR0.

Когда этот бит выставлен, привилегированный код может изменять данные только на страницах, на которые разрешена запись. Чтобы привилегированный код мог изменять данные на странице, бит R/W должен быть выставлен в каждом элементе, описывающем данную страницу, независимо от бита U/S.

2.3.6. Практика

Теперь пришло время применить все наши знания на практике. Мы реализуем обычный механизм трансляции адресов с размером страницы 4 Кб. Чтобы продемонстрировать главную возможность механизма трансляции адресов, создадим три страницы по разным адресам, которые будут спроецированы на один адрес (0B8000h) и, чтобы убедиться, что всё работает, выведем текст через эти адреса. Произведём эти действия в обычном режиме трансляции страниц и в режиме PAE. В обоих случаях будем располагать все таблицы страниц вместе в одной области памяти. В первом примере адрес нужной нам таблицы будет вычисляться так:

```
pta = pdi * 4096 + ptba
```

где: `pta` – адрес таблицы страниц, `pdi` – индекс в каталоге страниц, `ptba` – базовый адрес области памяти, где находятся все таблицы страниц.

Адрес каталога страниц в режиме PAE будет вычисляться так:

```
pda = pdti * 4096 + pdba
```

где: `pda` – адрес каталога страниц, `pdti` – индекс к таблице указателей на каталоги, `pdba` – базовый адрес области памяти, где находятся все каталоги страниц.

Адрес нужной нам таблицы страниц будет вычисляться так:

```
pta = pdti * pdi * 4096 + ptba
```

2.3.6.1. Обычный режим

Реализация примера обычного режима трансляции адресов, в отличие от режима PAE, не представляет особой сложности: есть всего один каталог страниц и 1024 таблицы страниц; для описания всех страниц достаточно 4 Мб памяти для таблиц страниц плюс одна страница для каталога. Как уже было сказано в предыдущих разделах, у нас есть готовый код, который переводит процессор в защищённый режим – надо только изменить файл PM_CODE.ASM.

В примере, который приведён ниже, самое главное – это код, который создаёт виртуальную страницу, т. е. необходимые структуры в каталоге страниц и в таблице страниц. Далее приводится код функции, которая принимает в регистре EAX виртуальный адрес, а в регистре EBX – физический.

Листинг 2.14a. Создание виртуальной страницы

```
create_PDEPTE:
; in
; EAX page address
; EBX phys page address
   pushad
```

```
mov edi, ebx
mov edx, eax

shr eax, 22
shl eax, 2
mov esi, PAGE_DIR_BASE_ADDRESS
add esi, eax
shr eax, 2
```

В начале функции мы сохраняем все регистры в стеке, чтобы потом их восстановить, а также виртуальный и физический адреса. Константа PAGE_DIR_BASE_ADDRESS хранит адрес каталога страниц. Потом мы получаем индекс в каталоге страниц (сдвиг вправо на 22 бита), умножаем его на 4 (битовый сдвиг на 2 влево – это то же самое, что и умножение на 4) и прибавляем его к адресу каталога страниц. В итоге мы получаем в регистре EAX адрес нужного нам элемента в каталоге страниц.

Листинг 2.14б. Создание виртуальной страницы *(продолжение)*

```
mov ebx, eax
shl ebx, 12

mov eax, PAGE_TABLES_BASE_ADDRESS
add eax, ebx
or eax, 011b
mov [esi], eax
and eax, 0FFFFF000h
```

В листинге 2.14б мы сначала умножаем смещение элемента в каталоге страниц на 4096 (4 Кб) и прибавляем к нему базовый адрес всех таблиц страниц (который содержит константа PAGE_TABLES_BASE_ADDRESS), и в итоге получаем адрес нужной нам таблицы страниц. Потом устанавливаем флаг присутствия и разрешения записи, сохраняем полученное значение в элементе каталога страниц (на него указывает регистр ESI, а значение находится в регистре EAX), потом сбрасываем установленные флаги в регистре EAX, и регистр EAX снова указывает на таблицу страниц.

Листинг 2.14в. Создание виртуальной страницы *(продолжение)*

```
.create_PTE:   ;eax = table base address
 mov esi, eax
 mov eax, edx
 shl eax, 10
 shr eax, 22
 shl eax, 2
 add esi, eax
```

В вышеприведённом листинге мы вначале заносим в регистр ESI указатель на таблицу страниц, который содержится в регистре EAX. Потом снова заносим виртуальный адрес в регистр EAX, который был сохранён в регистре EDX в начале функции. Потом получаем индекс в таблице страниц и умножаем его на 4

посредством битовых сдвигов. Наконец, добавляем полученное значение к адресу таблицы страниц, который хранится в регистре ESI. В результате всех этих манипуляций регистр ESI указывает на нужный нам элемент в таблице страниц.

Листинг 2.14г. Создание виртуальной страницы *(продолжение)*

```
    mov eax, edi
    or eax, 011b
    mov [esi], eax

  .end:
    popad
    ret
```

Вначале мы заносим физический адрес в регистр EAX, который был сохранён в регистре EDI в начале функции. Устанавливаем флаги присутствия и разрешения записи на страницу и сохраняем полученное значение в элементе таблицы страниц.

Написав функцию `create_PDEPTE`, мы можем с лёгкостью создать виртуальную страницу, что и показано в листинге 2.15.

Листинг 2.15. Использование функции create_PDEPTE

```
START_CODE:
    mov eax, START_CODE
    mov ebx, eax
    call create_PDEPTE

    mov eax, 0B8000h
    mov ebx, eax
    call create_PDEPTE

    mov eax, 0FF000000h
    mov ebx, 0B8000h
    call create_PDEPTE

    mov eax, 0EE000000h
    mov ebx, 0B8000h
    call create_PDEPTE
```

Следует отметить, что необходимо «создавать» виртуальные страницы не только для данных, но и для кода – ведь он тоже находится в памяти. Чтобы запустить механизм трансляции страниц, надо выставить самый старший бит в регистре CR0, но перед этим загрузить в регистр CR3 адрес каталога страниц (листинг 2.16).

Листинг 2.16. Включение механизма трансляции страниц

```
    mov eax, PAGE_DIR_BASE_ADDRESS
    mov cr3, eax

    mov eax, cr0
    or eax, 80000000h
    mov cr0, eax
```

Теперь после инициализации всех необходимых структур для механизма трансляции адресов осталось только вывести три сообщения по разным виртуальным адресам, которые проецированы на один физический (листинг 2.17).

Листинг 2.17. Демонстрация работы механизма трансляции страниц

```
    mov esi, message1
    mov edi, 0B8000h
    mov ecx,message2-message1
    rep movsb

    mov esi, message2
    mov ecx, message3-message2
    rep movsb

    mov esi, message3
    mov ecx, end_messages-message3
    rep movsb

    jmp    $

message1 db "152535455565758595 5 5"
message2 db "A5d5r5F5F5050505050505 5"
message3 db "A5d5r5E5E5050505050505 5"
end_messages:
```

Если мы видим строку `123456789 AdrFF000000 AdrEE000000` на экране после запуска программы, значит, всё нормально.

2.3.6.2. Режим PAE

Режим PAE более сложен в реализации, т. к. для описания одной страницы требуется больше структур, чем в обычном режиме. Пример реализации почти не отличается от предыдущего – за исключением функции создания виртуальной страницы. Поэтому далее приведён только код этой функции.

Листинг 2.18a. Создание виртуальной страницы PAE

```
create_VirtAddressPAE:
; in
; EAX page address
; EBX phys page address
   pushad

   and eax, 0FFFFF000h
   and ebx, 0FFFFF000h

   mov edi, eax
   mov esi, ebx
```

В начале функции происходит сброс 12 младших бит виртуального и физического адреса и сохранение их в других регистрах.

Листинг 2.18б. Создание виртуальной страницы PAE *(продолжение)*

```
shr eax, 30
mov ebx, eax
mov ecx, eax
shl eax, 3
add eax, PAGE_DIRECTORIES_POINTERS_TABLE_BASE_ADDRESS
     ; eax point to entry of Page Directories Table
```

Мы сначала получаем индекс в таблице указателей на каталоги. Потом сохраняем этот значение в регистрах EBX и ECX. Потом регистр EAX умножаем на 8. Прибавляем к полученному значению адрес таблицы указателей на каталоги и в итоге получаем в регистре EAX указатель на элемент в таблице указателей на каталоги.

Листинг 2.18в. Создание виртуальной страницы PAE *(продолжение)*

```
shl ebx, 12
add ebx, PAGE_DIRECTORIES_TABLE_BASE_ADDRESS
     ; ebx point to Page Directory

push ebx
or ebx, 1
mov [eax], ebx
pop ebx
```

Вначале мы умножаем индекс в таблице указателей на каталоги на 4096 и прибавляем к нему базовый адрес всех каталогов страниц – в результате получаем указатель на нужный нам каталог страниц. Сохраняем его в стеке, устанавливаем флаг присутствия в этом значении и заносим его в элемент таблицы указателей на каталоги, а потом восстанавливаем искомый адрес из стека.

Листинг 2.18г. Создание виртуальной страницы PAE *(продолжение)*

```
mov eax, edi
shl eax, 2
shr eax, 23
mov edx, eax
shl eax, 3
add eax, ebx
```

Снова заносим значение виртуального адреса в регистр EAX. Получаем из него индекс в каталоге страниц. Сохраняем полученный индекс в регистре EDX. Умножаем регистр EAX на 8. Потом прибавляем его к адресу каталога страниц, который содержится в регистре EBX. В результате получаем указатель на нужный нам элемент в каталоге страниц.

Листинг 2.18д. Создание виртуальной страницы PAE *(продолжение)*

```
imul edx, ecx
shl edx, 12
add edx, PAGE_TABLES_BASE_ADDRESS
```

```
push edx
or edx, 3
mov [eax], edx
pop edx
```

В листинге 2.18д мы сначала умножаем индекс в каталоге страниц и индекс в таблице указателей на каталоги. Произведение умножаем на 4096 и прибавляем к нему базовый адрес всех таблиц страниц. В результате получаем указатель на нужную нам таблицу страниц. Потом сохраняем регистр EDX в стеке, устанавливаем флаги присутствия и разрешения записи в адресе, заносим это значение в элемент каталога страниц. После всех манипуляций восстанавливаем адрес таблицы страниц из стека.

Листинг 2.18е. Создание виртуальной страницы PAE *(продолжение)*

```
mov ebx, edi
shl ebx, 11
shr ebx, 23
shl ebx, 3
add ebx, edx

or esi, 3
mov [ebx], esi
.end:
popad
ret
```

В завершающей части процедуры (листинг 2.18е) заносим виртуальный адрес в регистр EBX, получаем из него адрес индекс в таблице страниц. Умножаем его на 8 и прибавляем к нему адрес таблицы страниц; в итоге регистр EBX будет указывать на нужный элемент в таблице страниц. В конце концов мы выставляем флаги присутствия и разрешения записи в физическом адресе, который мы сохранили в регистре ESI в самом начале функции, и сохраняем полученное значение в элементе таблицы страниц.

Чтобы включить режим PAE, надо сначала загрузить в регистр CR3 адрес таблицы указателей на каталоги страниц, потом установить бит PAE в регистре CR4 и только потом включить режим трансляции страниц. Ниже приведён пример включения режима PAE.

Листинг 2.19.Включение режима PAE

```
mov eax, PAGE_DIRECTORIES_POINTERS_TABLE_BASE_ADDRESS
mov cr3, eax

mov eax, cr4
or eax, 32
mov cr4, eax

mov eax, cr0
or eax, 80000000h
mov cr0, eax
```

Полные исходые коды обеих программ находятся в папке part2 на компакт-диске, прилагающемся к данной книге.

2.3.6.3. Использование страниц разного размера

Операционная система одновременно может использовать страницы размером 4 Кб и 4 Мб. Наиболее наглядный пример совместного использования страниц разного размера – это использование больших страниц для адресации памяти ядра операционной системы.

Использование страниц большого размера увеличивает производительность процессора, т. к. он расходует меньше времени на трансляцию адреса. Использование страниц большого размера увеличит скорость работы ядра системы.

Использование страниц малого размера позволит приложениям эффективнее использовать память, уменьшив гранулярность её выделения. Например, если приложению требуется память размером 48 Кб, то при использовании больших страниц бесполезно будет занята память размером 4048 Кб, а при использовании малых страниц (всего 12 страниц) – лишь столько, сколько нужно программе.

Всё вышесказанное относится как к обычному режиму трансляции страниц, так и к режиму PAE.

2.3.7. Резюме

В этом разделе мы изучили механизм страничного преобразования: обычный режим и режим расширенной физической адресации (PAE). В следующем разделе речь пойдёт о многозадачности в защищённом режиме.

2.4. Многозадачность

Многозадачность – это один из основных параметров всех современных операционных систем. Основная единица измерения в многозадачной системе – это задача, или поток (в дальнейшем – задача). *Задача* – это некоторая самостоятельная последовательность команд (программа), которая выполняется в своём окружении. Основные параметры, которыми характеризуется окружение задачи, – это состояние регистров общего назначения, состояние сегментных регистров и адресное пространство (если операционная система обеспечивает изолированность задач друг от друга), которое характеризуется регистром CR3, а также состоянием регистров математического сопроцессора.

Поскольку в большинстве случаев количество потоков намного больше количества процессоров, многозадачность реализуется путём быстрого переключения задач. Именно благодаря этому создаётся ощущение, что все задачи работают одновременно.

2.4.1. Общие сведения

Многозадачность на процессорах x86 реализуется аппаратно. Это означает, что не программа, а сам процессор (!) выполняет переключение задач, т. е. сохраняет состояние прерываемой задачи в специально отведённой области памяти и запускает новую, предварительно восстановив состояние новой задачи и такой же области

памяти. Область памяти, куда процессор сохраняет состояние задачи, называется *сегментом состояния задачи*, или TSS (Task State Segment).

У каждой задачи может быть 4 стека для каждого из уровней привилегий. Для каждой задачи может использоваться своё виртуальное адресное пространство, т. е. у каждой задачи может быть свой каталог страниц, никак не зависящий от других, и это позволяет обеспечить изолированность каждой задачи от других и от памяти системы и тем самым повысить защищённость операционной системы.

Для каждой задачи может быть применена *карта разрешения ввода-вывода*. Она может запретить или разрешить доступ к каждому порту ввода-вывода. Каждая задача помимо глобальной дескрипторной таблицы может использовать свою таблицу, локальную дескрипторную таблицу (LDT), в которой также могут содержаться дескрипторы (но не всех типов). Задачи бывают 16-битные и 32-битные. 16-битные задачи появились в процессорах 80286, а на процессорах 80386 и позже были уже 32-битные задачи. Но тем не менее 16-битные задачи были оставлены на современных процессорах для обеспечения обратной совместимости. В данном разделе речь пойдёт только о 32-битных задачах.

2.4.2. Сегмент задачи (TSS)

Для описания каждой задачи используется область памяти, где хранятся все её параметры (как минимум состояние её регистров). Как уже было сказано, эта область памяти называется *сегментом состояния задачи*, или Task State Segment (далее – TSS). Как и любая другая область памяти, TSS описывается дескриптором в GDT, и у каждого TSS есть свой селектор. На рис. 2.25 приведён формат 32-битного TSS.

В TSS есть динамические и статические поля. Динамические поля обновляются при каждом переключении задач. Ниже приводится список динамических полей TSS:

- регистры общего назначения и состояние регистров EAX, ECX, EDX, EBX, ESP, EBP, ESI и EDI хранятся в соответствующих полях TSS;
- регистры сегментов и состояние регистров ES, CS, SS, DS, FS, GS также хранятся в одноимённых полях TSS;
- состояние регистра флагов;
- состояние регистра EIP хранится в одноимённом поле TSS и тоже обновляется при каждом переключении задач. Фактически это поле является самым важным в TSS, потому что если бы его не было, программа выполнялась бы с самого начала при каждом переключении;
- ссылка на предыдущую задачу нужна для обеспечения вложенности задач. В этой ссылке хранится селектор задачи, которая выполнялась перед текущей. Ссылка на предыдущую задачу хранится в самом первом поле TSS под названием Previous Task Link.

Статические поля не обновляются процессором автоматически – он только читает их. Статические поля следует задавать во время создания задачи. Приведём список статических полей:

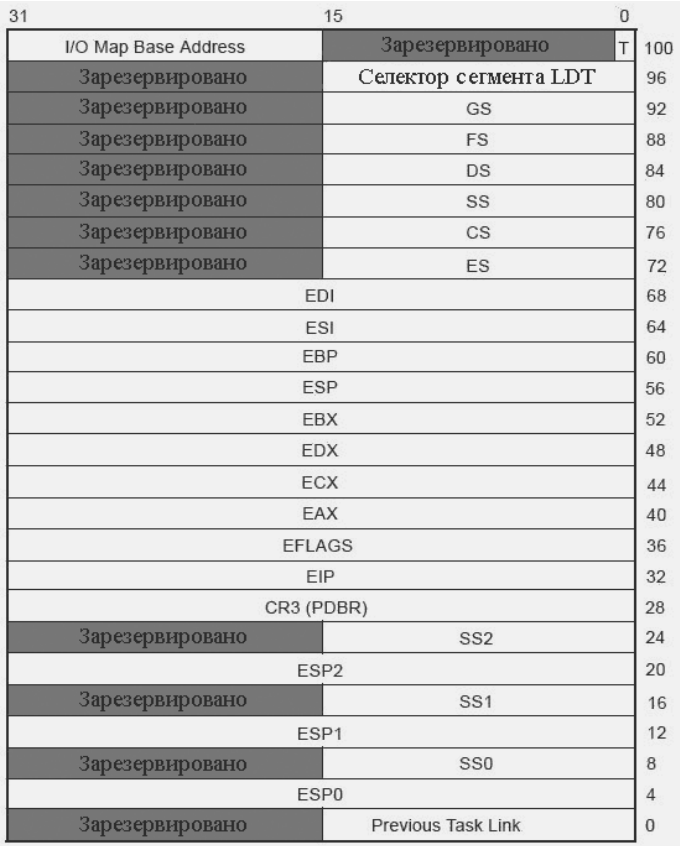


Рис. 2.25. Формат сегмента задачи

- селектор LDT;
- состояние регистра CR3;
- сегмент стека и указатель на стек для уровней привилегий 0, 1 и 2;
- флаг трассировки (бит 0 в байте со смещением 100) – если он выставлен, то процессор сгенерирует отладочное исключение сразу после переключения на эту задачу;
- 16-битное поле смещения карты разрешения ввода-вывода хранит смещение карты разрешения ввода-вывода относительно начала TSS.

Вышеприведённая схема TSS – это схема минимально допустимого TSS; если требуется хранить в нём дополнительные данные, такие как карта разрешения ввода-вывода, то размер TSS может быть увеличен. В TSS также можно хранить состояние других регистров процессора, но они уже не будут обрабатываться процессором – они должны обрабатываться программно.

Если включен механизм трансляции адресов, то TSS должен полностью располагаться в пределах одной страницы, потому что процессор преобразует

виртуальный адрес в физический только один раз. Если TSS находится на двух страницах и эти страницы отображены на несмежные физические страницы, то процессор может получить неверные данные о состоянии задачи.

2.4.3. Дескриптор TSS

Любой TSS должен быть описана в GDT специальным дескриптором. Дескриптор TSS может быть определён только в GDT. В целом структура дескриптора TSS сходна со структурой остальных типов дескрипторов, но есть и различия. На рис. 2.26 приведена схема дескриптора TSS.

Если флаг G сброшен, то поле лимита должно содержать значение не менее 67h (минимальный размер любого TSS составляет 68h; следовательно, минимальный предел 67h). В противном случае будет генерироваться исключение недопустимого TSS (`#TS`). Поскольку дескриптор TSS может находиться только в глобальной дескрипторной таблице, то второй бит селектора дескриптора TSS всегда должен быть сброшен.

Бит B (Busy) показывает занятость задачи; если он выставлен, то задача в настоящий момент выполняется. Это нужно для того, чтобы задача не могла переключаться на саму себя.

2.4.4. Локальная дескрипторная таблица

Локальная дескрипторная таблица (LDT, Local Descriptor Table) является аналогом глобальной дескрипторной таблицы GDT и предназначена для применения в контексте задачи. Таким образом, у задачи может быть свой собственный набор дескрипторов, непосредственный доступ к которым имеет только она сама.

Локальная дескрипторная таблица (далее – LDT) имеет сходную с GDT структуру. В LDT также нельзя использовать нулевой дескриптор. Главное отличие LDT от GDT заключается в том, что в ней нельзя определять дескрипторы системных объектов. *Системными объектами* являются те объекты, которые использует сам процессор. Таким объектом, например, является TSS. Это означает, что задача не может определять задачи в самой себе. В общей сложности в LDT можно определять только дескрипторы сегментов кода и данных и дескрипторы шлюзов. Это обеспечивает защищённость системы. Теперь вспомним формат селектора, который мы рассматривали в разделе 2.1.

Бит TI показывает, из какой таблицы будет браться дескриптор. Если он сброшен, то дескриптор берётся из GDT; если выставлен – то из LDT.

Рис. 2.26. Формат дескриптора сегмента задачи

Рис. 2.27. Селектор

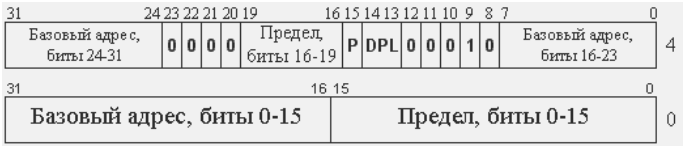


Рис. 2.28. Формат дескриптора LDT

LDT должна описываться в GDT специальным дескриптором. Формат дескриптора LDT приведён на рис. 2.28.

Чтобы получить возможность использовать дескрипторы из LDT, надо загрузить селектор дескриптора LDT в регистр LDTR. Это осуществляется командой LLDT – единственным операндом, которым является 16-битный регистр или значение памяти, содержащее селектор LDT (например, `lldt ax`). Наряду с командами LGDT и LIDT эта команда является привилегированной и может выполняться только на нулевом уровне привилегий. Для получения содержимого LDTR используется аналогичная команда SLDT; она не является привилегированной.

Регистр LDTR, как и все сегментные регистры, имеет теневую 8-байтовую часть, где содержится дескриптор, который использует только сам процессор. Это нужно для повышения производительности процессора, чтобы он не обращался каждый раз к глобальной дескрипторной таблице для выяснения лимита и адреса LDT. При изменении дескриптора в LDT, для того чтобы изменения вступили в силу, следует перезагрузить регистр LDTR.

Регистр LDTR обновляется каждый раз при переключении задачи.

2.4.5. Регистр задачи (TR)

Регистр TR (Task Register) – 16-разрядный регистр, который содержит селектор TSS текущей задачи. Он также имеет 8-байтовую теневую часть. Этот регистр нужен для двух целей:

1. Для определения текущей задачи в обработчиках прерываний, которые не являются задачами. Очень полезно при отладке.
2. Для перевода в режим многозадачности.

Загрузка регистра TR осуществляется командой LTR, а получение значения – командой STR; синтаксис аналогичен синтаксису команд LLDT/SLDT. Команда LTR является привилегированной, STR – непривилегированной. При загрузке регистра TR в дескрипторе TSS задачи устанавливается флаг занятости. Загрузка этого регистра связывает текущее состояние процессора с состоянием данной задачи.

2.4.6. Управление задачами

Переключение на задачу процессор производит автоматически и сам использует дескриптор TSS. Чтобы переключить задачу, достаточно выполнить всего одну из команд дальней передачи управления. Таких команд три:

1. JMP.
2. CALL.
3. IRET.

При переключении на задачу в качестве сегмента надо указать селектор TSS нужной задачи; в данном случае процессор будет игнорировать смещение. Если используется команда IRET, то селектор нужной задачи берётся из поля Previous Task Link текущей задачи.

Для того чтобы переключиться с одной задачи на другую, нужно, чтобы процессор уже выполнял какую-либо задачу, т. е. переключиться на задачу можно только из задачи. Это значит, что, во-первых, просто запустить первую задачу не получится – нужны некоторые хитрости, об этом мы поговорим позже. А во-вторых, из задачи нельзя «вернуться» в обычную процедуру, т.е. однажды запустив многозадачность, процессор остаётся в ней до перехода в режим реальных адресов или до выключения.

Осуществляя переключение между задачами, процессор использует несколько флагов:

- флаг занятости B находится в дескрипторе TSS, в шестом байте. Устанавливается всякий раз, когда происходит переключение на задачу (когда установлен, это 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 JMP либо IRET; при переключении командой CALL либо при прерывании (даже если обработчик прерывания – тоже задача) флаг не сбрасывается;
- флаг трассировки T находится в сегменте состояния задачи TSS, это нулевой бит по смещению 64h в TSS. Если флаг установлен, то при переключении на задачу процессор после успешной загрузки значений из всех полей TSS 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 вложенности NT (Nested Task) находится в регистре EFLAGS. Если переключение на новую задачу было вызвано командой 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 Previous Task Link в TSS текущей задачи;
- флаг TS (Task Switched) находится в регистре управления CR0. Этот флаг устанавливается каждый раз, когда процессор переключается на другую задачу, и служит индикатором переключения задач. При попытке выполнить команды FPU, MMX или XMM процессор может генерировать исключение отсутствующего устройства (#NM – прерывание 7), что позволяет системе выполнить смену контекста FPU, MMX и XMM.

Процессор производит переключение задач в следующих четырёх случаях:

1. При выполнении межсегментного перехода (JMP) или межсегментного вызова (CALL), когда в качестве селектора указан селектор TSS.
2. При выполнении межсегментного перехода (JMP) или межсегментного вызова (CALL), когда в качестве селектора указан селектор шлюза задачи в GDT или LDT.
3. При вызове обработчика прерывания или исключения, если ему в IDT соответствует шлюз задачи.
4. При выполнении команды IRET, когда EFLAGS.NT=1.

В процессе переключения с одной задачи на другую процессор выполняет следующие действия:

1. Получает селектор TSS из кода команды (случай 1), из шлюза задачи (случаи 2 и 3), из поля Previous Task Link текущей задачи (случай 4).
2. Выполняется контроль привилегий:
   a) максимальный из RPL и CPL должен быть не больше чем DPL дескриптора TSS;
   b) максимальный из RPL и CPL должен быть не больше чем DPL шлюза задачи (или DPL дескриптора TSS);
   c) в случаях 3 и 4 проверка не выполняется.
3. Выполняется контроль предела TSS.
4. Проверяется доступность задачи: в случаях 1–3 задача должна быть незанятой, в случае 4 (возврат из подзадачи) она должна быть помечена как занятая.
5. Выполняется проверка присутствия старого TSS, нового TSS и всех сегментов в новой задаче.
6. Если переключение инициируется JMP или IRET, для текущей задачи очищается флаг занятости (в поле типа дескриптора). В случае CALL или вызова обработчика прерывания/исключения флаг занятости в старой задаче остается установленным.
7. Если переключение инициируется IRET, для старой задачи сбрасывается бит вложенной задачи (NT) в регистре флагов. В остальных случаях значение EFLAGS.NT для старой задачи не меняется.
8. Процессор сохраняет контекст процессора в соответствующих полях сегмента состояния старой задачи.
9. Если переключение инициировано CALL или вызовом обработчика прерывания/исключения, в контексте новой задачи выставляется бит вложенной задачи (EFLAGS.NT=1), а в поле Previous Task Link новой задачи заносится селектор TSS старой задачи. В противном случае (JMP, IRET) бит остается без изменений.
10. Дескриптор TSS новой задачи помечается как занятый (либо остаётся помеченным как занятый).
11. В регистр задачи (TR) загружается новое значение.
12. Загружается контекст процессора из TSS новой задачи.
13. Начинается выполнение новой задачи.

2.4.7. Шлюз задачи

В некоторых случаях в качестве обработчика прерывания используется *шлюз задачи*. Про шлюз задачи мы вскользь говорили в разделе 2.2, посвящённом прерываниям; теперь, когда речь идёт о многозадачности, самое время их вспомнить.

Использование шлюза задачи дополнительно защищает и изолирует обработчики прерываний от обычных приложений и повышает надёжность системы. Шлюз задачи может находиться как в IDT, так и в GDT и в LDT. Формат дескриптора шлюза задачи представлен на рис. 2.29.

Как видно из рисунка, в шлюзе задачи есть только одно поле с конкретным значением – это селектор TSS, который указывает, на какую задачу произойдёт переключение при проходе через этот шлюз.

Рис. 2.29. Формат дескриптора шлюза задачи

2.4.8. Уровень привилегий ввода-вывода

Поле IOPL регистра флагов используется для контролирования ввода-вывода прикладных программ. Это 2-битовое поле задаёт текущий уровень привилегий ввода-вывода. Следующие инструкции могу быть выполнены, только если текущий уровень привилегий (CPL) меньше либо равен уровню привилегий ввода-вывода (CPL <= IOPL): in, ins, out, outs, cli, sti. Эти команды чувствительны к уровню привилегий ввода-вывода. В ответ на любую попытку менее привилегированной задачи использовать ввод-вывод чувствительная команда генерирует исключение общей защиты (#GP).

Программа или задача может изменить свой уровень привилегий ввода-вывода только с командами POPF и IRET; однако такие изменения привилегированы. Никакой код не может изменить текущий уровень привилегий ввода-вывода, если он не выполняется на нулевом уровне привилегий. Попытка менее привилегированного кода изменить уровень привилегий ввода-вывода не приводит к исключению; уровень привилегий ввода-вывода просто остается неизменным. Команда POPF также может использоваться, чтобы изменить состояние флага IF; однако команда POPF в этом случае также чувствительна к IOPL. Программа может использовать команду POPF для изменения флага IF, только если текущий уровень привилегий меньше или равен текущему уровню привилегий ввода-вывода. Попытка менее привилегированного кода изменить флаг IF не приводит к исключению – флаг IF остается неизменным.

2.4.9. Карта разрешения ввода-вывода

Карта разрешения ввода-вывода контролирует доступ к портам ввода-вывода менее привилегированных программ, т. е. тех, у которых текущий уровень привилегий больше, чем уровень привилегий ввода-вывода. Карту разрешения ввода-вывода можно задать для каждой задачи, и она находится в TSS. Первый байт карты ввода-вывода находится по адресу, который содержится в поле IOMapBaseAddress в TSS; оно содержит адрес относительно начала TSS. Адрес карты и размер могут быть любыми, главное – чтобы фактический размер TSS не превышал лимит указанный в дескрипторе TSS.

Если текущий уровень привилегий больше, чем уровень привилегий ввода-вывода, то процессор обращается к карте разрешения ввода-вывода, чтобы узнать, разрешить или запретить доступ к конкретному порту ввода-вывода. Каждый бит в карте отвечает за один 1-байтовый порт ввода-вывода. Если бит установлен, то доступ к соответствующему порту запрещен, если сброшен – разрешен. Например, бит контроля для порта ввода-вывода 29h (41) будет находиться в первом бите из шестого байта в карте.

Перед предоставлением доступа к порту ввода-вывода процессор проверяет все биты, соответствующие этому порту ввода-вывода. Например, для проверки возможности доступа к порту размером в двойное слово процессор проверяет все четыре бита, соответствующие четырем смежным 1-байтовым портам. Если хотя бы один проверяемый бит установлен, то генерируется исключение общей защиты (#GP). Если все проверяемые биты сброшены, то операция ввода-вывода разрешается.

Каждый раз при обращении к карте разрешения ввода-вывода процессор считывает сразу два байта. Во избежание генерации исключений после карты ввода-вывода должен быть байт, у которого все биты выставлены (0FFh).

Нет необходимости включать в карту разрешения ввода-вывода данные на все адреса портов ввода-вывода. Адреса ввода-вывода, не включённые в карту, рассматриваются как запрещённые. Карта разрешения ввода-вывода должна размещаться в конце TSS, поскольку именно предел TSS задаёт размер карты – т. е. где кончается TSS, там кончается и карта. Например, если лимит TSS на 10 байт больше адреса карты, то карта имеет размер 11 байт, и первые 80 портов ввода-вывода нанесены на карту. Обращение к адресам более 80 приводит к исключению общей защиты, т. е. они запрещены. Если адрес базы карты разрешения ввода-вывода больше или равен лимиту TSS, то считается, что карты разрешения ввода-вывода не существует, и все инструкции ввода-вывода генерируют исключения, когда CPL больше, чем IOPL.

2.4.10. Включение многозадачности

Как уже было сказано, просто так перейти в режим многозадачности нельзя. Это обусловлено тем, что переключиться на какую-нибудь задачу можно только из задачи. Для того чтобы перейти в режим многозадачности, надо произвести следующие действия:

1. Объявить сегменты и дескрипторы TSS в GDT.
2. Заполнить поля в TSS нужными значениями. Наиболее важными полями являются: EIP, CS, CR3 (если включён механизм страничного преобразования), SS, ESP.
3. Загрузить в регистр TR селектор TSS, который описывает текущий код.

После всех этих действий мы будем находиться в полноценном режиме многозадачности – можно спокойно определять новые задачи и переключаться на любую. Хотя не так важно, селектор какой именно TSS мы загрузим в TR: ведь какие бы ни были в ней значения, они всё равно заменятся параметрами текущей задачи при первом же переключении задач.

2.4.11. Практическая реализация

Теперь пришло время реализовать многозадачность на практике. У нас будет три задачи. Первая задача ничего не будет делать. А в подтверждение того, что вторая и третья задачи работают, они будут выводить сообщение. Также у нас будут включены прерывания, и на нулевом векторе будет стоять в качестве обработчика шлюз задачи. Именно этот обработчик и будет выполнять переключение между ними (второй и третьей задачей) по очереди. Суть работы обработчика будет заключаться в следующем: при получении управления он выясняет, что находится в Previous Task Link его TSS, и меняет значение селектора: если селектор второй задачи, то идёт замена на селектор третьей, и наоборот. Таким образом, обработчик прерывания будет прерывать одну задачу, а возвращаться уже в другую.

Начнём сначала. В первую очередь надо определить дескрипторы всех используемых нами TSS. Для этого следует добавить в конец GDT новые дескрипторы и обновить поле `Limit` в регистре GDTR. Но мы поступим иначе: скопируем GDT в другое место и добавим к ней новые дескрипторы, а затем полностью обновим регистр GDTR. Такой способ не является оптимальным – он просто нужен для того, чтобы переместить GDT за пределы первого мегабайта памяти. Не следует считать это обязательным требованием; это делается просто для разнообразия. Для более удобной работы с образом регистра GDTR можно объявить следующую структуру:

```
struc GDTR32
    {
    .Limit dw ?
    .BaseAddress dd ?
    }
```

Теперь достаточно объявить переменную и использовать её в командах L/SGDT.

```
GDTR_Image  GDTR32
```

Далее в листинге 2.20 приведён код, обновляющий глобальную дескрипторную таблицу.

Листинг 2.20. Обновление GDT

```
mov esi, message1
mov al, 0
mov ah, 0
```

```
mov bl, "5"
call OutText

xor eax, eax
mov edi,NewGDT
mov ecx, NewGDTSize/4
rep stosd

sgdt [GDTR_Image]
mov esi, [GDTR_Image.BaseAddress]
mov edi, NewGDT
xor ecx, ecx
inc word [GDTR_Image.Limit]
mov cx, [GDTR_Image.Limit]
shr ecx, 2
rep movsd

mov edi, NewGDT
mov ax, [GDTR_Image.Limit]
add edi, eax
mov esi, TssDescriptors
mov ecx, TssDescriptorsSize/4
rep movsd
```

Вначале мы выводим сообщение о том, что мы перешли в защищённый режим. Код функции `OutText` приводиться здесь не будет (это не особенно важно), но следует сказать о параметрах, которые она принимает: ESI – указатель на строку, которая заканчивается символом с кодом 0, AL – номер столбца (нумерация с нуля), AH – номер строки (нумерация с нуля), BL – параметры символов (цвет и фон). Далее память, куда будет копироваться GDT, обнуляется; это необязательно, но для надежности не помешает. Далее выполняется команда SGDT для получения параметров GDT. После этого она копируется по адресу NewGDT. Следует отметить, что GDTR содержит не размер таблицы, а её лимит (т. е. максимальное смещение в таблице), поэтому её лимит увеличивается на единицу. Затем производится копирование дескрипторов TSS в конец новой глобальной дескрипторной таблицы. Для объявления дескрипторов используется следующий макрос (листинг 2.21).

Листинг 2.21. Макрос, создающий дескриптор TSS

```
macro DEFINE_TSS_DESCRIPTOR _base_address,_limit
   {
   dw  _limit and 0FFFFh
   dw _base_address and 0FFFFh
   db (_base_address shr 16) and 0FFh
   db TSS_TYPE
   db (_limit shr 16) and 0Fh
   db _base_address shr 24
   }
```

Этот макрос принимает два параметра: адрес TSS и его лимит. Теперь, чтобы объявить все четыре дескриптора TSS, достаточно написать строки, которые вы видите в листинге 2.22.

Листинг 2.22. Объявление необходимых дескрипторов с помощью макроса DEFINE_TSS_DESCRIPTOR

```
TssDescriptors:
 DEFINE_TSS_DESCRIPTOR FirstTaskTss, 100h
 DEFINE_TSS_DESCRIPTOR TASK_NUMBER_1_Tss, 100h
 DEFINE_TSS_DESCRIPTOR TASK_NUMBER_2_Tss, 100h
 DEFINE_TSS_DESCRIPTOR Irq0_handler_Tss, 100h
TssDescriptorsEnd:
```

Настало время обновить GDTR и вывести сообщение об этом (листинг 2.23).

Листинг 2.23. Обновление регистра GDTR

```
 mov [GDTR_Image.BaseAddress], NewGDT
 mov ax, [GDTR_Image.Limit]
 add ax, TssDescriptorsSize
 dec ax
 mov [GDTR_Image.Limit], ax
 lgdt [GDTR_Image]

 mov esi, message2
 mov al, 0
 mov ah, 1
 mov bl, "5"
 call OutText
```

Теперь очередь заполнения полей каждого TSS. Для облегчения доступа к полям TSS используется структура, приведённая в листинге 2.24.

Листинг 2.24. Структура TSS32

```
struc TSS32
  {
   .PreviousTaskLink dw ?
   .Reserved0 dw ?    ; --4

   .ESP0       dd ?
   .SS0        dw ?
   .Reserved1  dw ?   ; --12

   .ESP1       dd ?
   .SS1        dw ?
   .Reserved2  dw ?   ; 20

   .ESP2       dd ?
   .SS2        dw ?
   .Reserved3 dw ?    ; 28
```

```
    .tsCR3      dd ?
    .tsEIP      dd ?   ; 36
    .tsEFLAGS   dd ?
    .tsEAX      dd ?   ; 44

    .tsECX      dd ?
    .tsEDX      dd ?   ; 52
    .tsEBX      dd ?
    .tsESP      dd ?   ; 60

    .tsEBP      dd ?
    .tsESI      dd ?
    .tsEDI      dd ?   ;72

    .tsES       dw ?
    .Reserved4  dw ?   ; 76
    .tsCS       dw ?
    .Reserved5  dw ?
    .tsSS       dw ?
    .Reserved6  dw ?
    .tsDS       dw ?
    .Reserved7  dw ?
    .tsFS       dw ?
    .Reserved8  dw ?
    .tsGS       dw ?
    .Reserved9  dw ?

    .LDTSegmentSelector dw ?
    .Reserved10 dw ?
    .DebugByte  db ?
    .Reserved11 db ?
    .IOMapBaseAddress dw ?
  }
```

Чтобы задать начальное состояние каждой задачи, можно использовать следующую функцию (листинг 2.25).

Листинг 2.25. Функция инициализации TSS

```
Set_TSS:
;IN
; EAX  - CR3
; EBX  - EIP
; ECX  - ESP
; EDX  - EFLAGS
; EDI  - TSS base address
     pushad

     push eax
     push edi
     push ecx
```

```
xor eax, eax
mov ecx, 26
cld
rep stosd

pop ecx
pop edi
pop eax

virtual at edi
.edi TSS32
end virtual

mov [.edi.tsCR3], EAX
mov [.edi.tsEIP], EBX
mov [.edi.tsESP], ECX
mov [.edi.tsEFLAGS], EDX

mov [.edi.tsCS], CODE_SELEKTOR
mov [.edi.tsDS], DATA_SELEKTOR
mov [.edi.tsSS], DATA_SELEKTOR
mov [.edi.tsES], DATA_SELEKTOR

popad
ret
```

Эта функция принимает пять параметров любой задачи (наиболее важных для неё) через регистры общего назначения: EAX – CR3, EBX – EIP, ECX – ESP, ; EDX – EFLAGS, EDI – адрес TSS. Сначала функция обнуляет первые 104 байта TSS (это нужно для того, чтобы в тех регистрах, которые мы не задали, гарантированно были нулевые значения). Здесь мы использовали директиву компилятора `virtual at`. Эта директива нужна для облегчения доступа к структурам через метки и регистры, которые на них указывают. Например, mov [.edi.tsCR3], EAX в результате будет заменена на mov [edi+28], EAX. Возвращаемся к нашему примеру – надо задать начальные параметры каждой задачи. Используя функцию `Set_TSS`, сделать это очень легко (листинг 2.26).

Листинг 2.26. Инициализация задач

```
mov eax, CR3  ; EAX = CR3
pushfd
pop edx       ; EDX = EFLAGS
mov ecx, Main_task_stack
mov ebx, FirstTask
mov edi, FirstTaskTss
call Set_TSS

mov ecx, Task_number1_Stack
mov ebx, TASK_NUMBER_1
```

```
    mov edi, TASK_NUMBER_1_Tss
    call Set_TSS

    mov ecx, Task_number2_Stack
    mov ebx, TASK_NUMBER_2
    mov edi, TASK_NUMBER_2_Tss
    call Set_TSS

    mov ecx, irq0_handler_Stack
    mov ebx, irq0_handler
    mov edi, Irq0_handler_Tss
    call Set_TSS
```

Код инициализации контроллеров прерываний и IDT здесь приводиться не будет. Теперь для включения многозадачности достаточно загрузить в регистр TR селектор первой задачи (листинг 2.27).

Листинг 2.27. Включение многозадачности

```
    mov ax, FirstTask_Selektor
    ltr ax

FirstTask:

    mov esi, message3
    mov al, 0
    mov ah, 2
    mov bl, "5"
    call OutText

    sti
    jmp $
```

Следует отметить, что метка FirstTask, по сути дела, вообще не нужна – она введена лишь для улучшения «читабельности» кода. В первой задаче мы просто выводим сообщение о том, что многозадачность включена, разрешаем прерывания и уходим в вечный цикл.

Код обоих задач почти одинаков; в листинге 2.28 приведён код первой задачи.

Листинг 2.28. Код первой задачи

```
message5 db "Task number 1 message  "
counter1 dd 11111111h
      db 0
TASK_NUMBER_1:
     sti

     mov esi, message5
     mov al, 0
```

```
    mov ah, 6
    mov bl, "5"

    @@:
    call OutText

    inc dword [counter1]
    jmp @b
```

Задача включает прерывания и уходит в вечный цикл вывода текста. Причина повторного включения прерываний заключается в том, что регистр флагов является копией регистра флагов первой задачи ещё до того, как многозадачность была включена, а тогда флаг IF был сброшен и, следовательно, прерывания запрещены.

При каждом повторении значение переменной `counter1` увеличивается на единицу и при выводе после текста «Task number 1 message» должны стоять четыре мелькающих символа. Метки под названием `@@` являются безымянными и используются совместно с директивами `@b` и `@f`; директива `@b` обозначает первую предыдущую метку, а `@f` – первую последующую метку.

Теперь осталось только привести код обработчика прерывания IRQ0 (листинг 2.29а).

Листинг 2.29а. Обработчик прерывания IRQ0

```
irq0_handler:

   mov edi, Irq0_handler_Tss

   virtual at edi
   .edi TSS32
   end virtual

  .repeat:
   inc dword [counter]

   mov esi,interrupt_mes
   mov al, 0
   mov ah, 4
   mov bl, "5"
   call OutText
```

Начало обработчика должно быть понятно. Заносим в регистр EDI указатель на TSS обработчика. В начале тела цикла мы выводим сообщение о том, что произошёл вызов обработчика (листинг 2.29б).

Листинг 2.29б. Обработчик прерывания IRQ0 *(продолжение)*

```
    xor eax, eax

    mov ax, [.edi.PreviousTaskLink]
```

```
call ClearBusyFlag

cmp ax, TASK_NUMBER_2_Selektor
jz @f
mov ax, TASK_NUMBER_2_Selektor
call SetBusyFlag
mov [.edi.PreviousTaskLink], TASK_NUMBER_2_Selektor

jmp .end_dispatch
@@:
mov ax, TASK_NUMBER_3_Selektor
call SetBusyFlag
mov [.edi.PreviousTaskLink], TASK_NUMBER_3_Selektor
```

Здесь мы производим замену селектора в поле Previous Task Link. Но для начала сбрасываем флаг занятости в дескрипторе предыдущей задачи с помощью функции `ClearBusyFlag`, она принимает в качестве параметра селектор нужного нам дескриптора в регистре AX. Код этой функции здесь приводиться не будет – его можно посмотреть в полном исходном коде примера, который есть на компакт-диске, прилагающемся к книге. Для установки флага занятости используется аналогичная функция `SetBusyFlag`. Если в поле `Previous Task Link` содержится селектор второй задачи, то заносим в него селектор третьей, в противном случае – селектор второй.

Листинг 2.29в. Обработчик прерывания IRQ0 *(продолжение)*

```
.end_dispatch:
mov  al, 20h
out  020h, al
out  0a0h, al

iretd
jmp .repeat
```

В конце обработчика (листинг 2.29в) мы производим операцию EOI и возвращаем управление командой IRETD. Следует отметить, что при следующем вызове обработчика прерывания выполнение обработчика продолжится с прерванного места, т. е. выполнится команда, следующая за командой IRETD, и эта команда вернёт нас в начало цикла.

Полный исходный код программы находится в папке part2 на компакт-диске, прилагающемся к данной книге.

2.4.12. Резюме

В этом разделе нами были изучены механизмы, обеспечивающие многозадачность в защищённом режиме. Тем не менее, как это ни прискорбно, наиболее распространённые операционные системы (Windows и UNIX) по каким-то причинам не используют аппаратный механизм поддержки многозадачности, а вместо этого реализуют многозадачность программно. В разделе 2.5 мы будем изучать механизмы защиты в защищённом режиме.

2.5. Механизмы защиты

Защищённый режим для операционной системы предоставляет широкий набор защитных механизмов. Когда используется механизм защиты, каждое обращение к памяти должно удовлетворять целому ряду условий защиты. Все проверки выполняются до начала обращения к памяти, и любые нарушения приводят к генерации исключений. Эти проверки происходят одновременно с трансляцией адреса и не приводят к потере производительности.

Установка флага PE в регистре CR0 заставляет процессор переключиться в защищённый режим, который в свою очередь автоматически обеспечивает механизм защиты. Ту часть, которая отвечает за защиту по привилегиям, можно отключить, если программа, работающая в защищённом режиме, будет использовать все дескрипторы и селекторы нулевого уровня привилегий.

Защита на уровне страниц автоматически обеспечивается при включении страничного преобразования установкой бита PG в CR0. Эту защиту можно отключить, сбросив флаг WP в регистре CR0 и установив флаги R/W и (U/S) в каждом элементе каталога и таблиц страниц. В результате каждая страница становится доступной для чтения и записи с любого уровня привилегий, что отменяет защиту на уровне страниц.

2.5.1. Поля и флаги, используемые для защиты на уровне сегментов и страниц

Флаг типа дескриптора (S) – 12-й бит во втором двойном слове дескриптора. Определяет вид дескриптора: если 0, то он описывает системный объект, если 1 – то сегмент кода или данных.

Поле типа дескриптора – биты 8–11 второго двойного слова дескриптора. Определяют тип системного объекта, сегмента кода или данных.

Поле предела дескриптора – биты 0–15 и 48–51 в дескрипторе. Определяют размер сегмента в единицах размера: байтах или 4-килобайтовых страницах (в зависимости от флага G).

Флаг гранулярности (G) – бит 23 второго двойного слова дескриптора. Определяет единицы измерения размера сегмента: 0 – сегмент измеряется в байтах, 1 – в 4-килобайтовых страницах.

Флаг E – бит 10 второго двойного слова дескриптора сегмента данных. Определяет расширение сегмента: 0 – адреса растут вверх (как в обычном сегменте), 1 – вниз (т. е. в обратном порядке, как в стеке).

Поле уровня привилегий дескриптора (DPL) – биты 13 и 14 второго двойного слова любого дескриптора. Определяют уровень привилегий объекта.

Поле запрашиваемого уровня привилегий (RPL) – биты 0 и 1 любого селектора. Содержат запрашиваемый уровень привилегий при обращении к дескриптору.

Поле текущего уровня привилегий (CPL) – биты 0 и 1 регистра CS. Содержат уровень привилегий текущего выполняемого кода.

Флаг пользователя/системы (U/S) – бит 2 элемента каталога или таблицы страниц. Определяет тип страницы: 0 – системный, 1 – пользовательский.

Поле чтения/записи (R/W) – бит 1 элемента каталога или таблицы страниц. Определяет тип доступа к странице: 0 – только чтение, 1 – чтение и запись.

Как показала практика, в самой распространённой операционной системе защищённого режима Windows защита на уровне сегментов почти отсутствует. Всем программам предоставляются сегменты, покрывающие всё доступное адресное пространство. При этом защита данных операционной системы полностью возлагается на механизм трансляции страниц.

2.5.2. Проверка лимитов сегментов

Проверка предела дескриптора сегмента не позволяет программе обращаться за пределы сегмента. Значение предела зависит от флага гранулярности дескриптора. Предел определяет максимальное значение смещения в сегменте.

Непонятным остаётся только, как проверяется предел у сегментов, расширяющихся вниз. Для сегментов данных, расширяющихся вниз, предел определяет последний адрес, доступ к которому запрещён внутри сегмента. Допустимыми будут адреса в диапазоне от (предел + 1) до FFFFh, если флаг D=0, и в диапазоне от (предел + 1) до FFFFFFFFh, если D=1. Максимальный размер такие сегменты имеют с пределом равным нулю. Помимо проверок обычных сегментов, процессор также проверяет пределы дескрипторных таблиц GDT, IDT, LDT и текущего сегмента TSS, не позволяя обращаться за их пределы.

2.5.3. Проверки типов

Дескриптор сегмента содержит информацию о типе в двух элементах:

- флаг S (тип дескриптора);
- поле типа.

Процессор использует эту информацию для определения ошибочных действий программ, когда они пытаются использовать сегмент или шлюз неправильным или несоответствующим образом.

Флаг S определяет, что описывает дескриптор: системный объект или сегмент кода либо данных. Поля 9–11 содержат 4 дополнительных бита, определяющие различные типы несистемных дескрипторов. В табл. 2.3 приведены всевозможные комбинации флагов в несистемных дескрипторах.

Бит 8 содержит бит A – он обозначает, был ли произведён доступ к данному сегменту с момента последнего сброса этого бита. Сегменты кода могут быть согласованными и несогласованными. Согласованность или несогласованность сегмента определяет флаг C (бит 10, во втором двойном слове дескриптора). Про это мы ещё поговорим позже. Сегмент, селектор которого указан в регистре SS, обязательно должен быть доступным к записи. Следует отметить, что сегмент стека не должен быть расширяющимся вниз; например, в системах Windows для стека используется обычный сегмент данных. В примерах к предыдущим разделам мы также использовали обычный сегмент данных в качестве сегмента стека.

Таблица 2.3. Флаги в несистемных дескрипторах

Биты			Тип
11	10	9	
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, выполнение и считывание
1	1	0	Код, только выполнение, согласованный
1	1	1	Код, выполнение и считывание, согласованный

В табл. 2.4 приведены всевозможные комбинации полей типа для системных объектов.

Таблица 2.4. Типы системных дескрипторов

Содержимое битов					Описание
Значение	11	10	9	8	
0	0	0	0	0	Зарезервировано
1	0	0	0	1	16-разрядный TSS (свободный)
2	0	0	1	0	LDT
3	0	0	1	1	16-разрядный TSS (занятый)
4	0	1	0	0	16-разрядный шлюз вызова
5	0	1	0	1	Шлюз задачи
6	0	1	1	0	16-разрядный шлюз прерывания
7	0	1	1	1	16-разрядный шлюз ловушки
8	1	0	0	0	Зарезервировано
9	1	0	0	1	32-разрядный TSS (свободный)
10	1	0	1	0	Зарезервировано
11	1	0	1	1	32-разрядный TSS (занятый)
12	1	1	0	0	32-разрядный шлюз вызова
13	1	1	0	1	Зарезервировано
14	1	1	1	0	32-разрядный шлюз прерывания
15	1	1	1	1	32-разрядный шлюз ловушки

Процессор проверяет значение типа несколько раз, когда оперирует селекторами и дескрипторами. Далее приведены примеры типичных операций, где происходит проверка типов (список не полный):

- в регистр CS можно загрузить только селектор сегмента кода;
- селектор сегмента нечитаемого кода нельзя загружать в сегментные регистры данных (DS, ES, FS и GS);

- в регистр SS можно загружать только селектор сегмента данных, разрешённого для записи;
- в LDTR можно загрузить только селектор для LDT;
- в TR можно загрузить только селектор для TSS;
- нельзя записывать в сегмент кода;
- нельзя записывать в сегмент данных, предназначенный только для чтения;
- нельзя читать из сегмента кода, предназначенного только для выполнения;
- команда LAR должна обращаться к дескриптору сегмента или шлюза для сегментов LDT, TSS, шлюза вызова, шлюза задачи, сегмента кода или данных;
- команда LSL должна обращаться к дескриптору сегмента LDT, TSS, кода или данных;
- элементом IDT могут быть только шлюзы прерывания, ловушки или вызова;
- при выполнении команд FAR CALL и FAR JMP процессор проверяет тип дескриптора, селектор которого содержится в адресе назначения этих команд. Если это дескриптор сегмента кода или шлюза вызова, то происходит передача управления по этому адресу; если дескриптор описывает TSS или шлюз задачи, то происходит переключение задач. (Проверки типа при переключении задач были указаны в разделе 2.4; проверки типа при передаче управления будут указаны далее.)

Попытка загрузки селектора нулевого дескриптора в регистр CS или SS приводит к генерации исключения общей защиты (#GP). Селектор нулевого дескриптора можно загружать в регистры DS, ES, FS или GS, но тогда обращение через них к памяти приводит к генерации исключения общей защиты.

2.5.4. Уровни привилегий

Механизм защиты защищённого режима использует четыре уровня привилегий – от 0 до 3. Меньшее значение соответствует большим привилегиям. Иногда эти уровни привилегий интерпретируются как кольца защиты (например, в системах Windows).

Текущий уровень привилегий – CPL. CPL – это уровень привилегий текущего исполняемого кода. Он хранится в битах 0 и 1 регистров CS и SS. Обычно CPL равен уровню привилегий сегмента кода, из которого выбираются команды. Процессор меняет CPL, когда управление передаётся сегменту кода с другим уровнем привилегий. CPL интерпретируется немного иначе, когда управление передаётся подчинённому сегменту кода. Подчинённый сегмент кода доступен с тех уровней привилегий, которых численно не меньше, чем DPL подчинённого сегмента кода. CPL не меняется, когда происходит передача управления на подчинённый сегмент кода, имеющий уровень привилегий, отличный от CPL.

Запрашиваемый уровень привилегий – RPL. RPL – это уровень привилегий, назначаемый селекторам сегментов. Он хранится в битах 0 и 1 селектора сегмента. Процессор проверяет RPL совместно с CPL, определяя возможность доступа к сегменту. Даже если программа или задача, запрашивающая доступ к сегменту, имеет достаточные привилегии для доступа, доступ будет запрещён, если RPL не имеет достаточных привилегий. Таким образом, если RPL селектора сегмента численно

превосходит CPL, то значение RPL будет решающим, и наоборот. RPL можно использовать для того, чтобы привилегированный код не мог обратиться к сегменту от имени прикладной программы, пока эта программа не получит достаточных на то привилегий.

Уровень привилегий дескриптора – DPL. DPL – это уровень привилегий сегмента или шлюза. Он хранится в поле DPL дескриптора сегмента или шлюза. При попытке доступа из текущего сегмента кода к другому сегменту кода или шлюзу DPL целевого дескриптора сравнивается с CPL и RPL. В зависимости от типа сегмента или шлюза DPL интерпретируется следующим образом:

- сегмент данных. DPL определяет наибольший номер уровня привилегий, который программа или задача может иметь для доступа к этому сегменту. Например, если DPL сегмента данных равен 1, то только программы, работающие на уровнях 0 и 1 (т. е. имеющие CPL равный 0 или 1), могут обращаться к этому сегменту;
- неподчинённый сегмент кода без использования шлюза вызова. DPL задаёт уровень привилегий, который должна иметь программа или задача для доступа к этому сегменту. Например, если DPL неподчинённого сегмента кода равен нулю, то только программа, работающая только на CPL равном нулю, может обратиться к этому сегменту;
- шлюз вызова. DPL определяет номер наибольшего уровня привилегий, который может иметь текущая программа или задача для доступа к этому шлюзу вызова (правило доступа – такое же, как и для сегмента данных);
- подчинённый или неподчинённый сегмент кода, доступный через шлюз вызова. DPL определяет наименьший номер уровня привилегий, который должна иметь программа или задача для доступа к этому сегменту. Например, если DPL подчинённого сегмента кода равен 2, то к нему будут иметь доступ только программы с CPL равным 2 или 3;
- TSS. DPL определяет наибольший номер уровня привилегий, с которого программа или задача может обратиться к этому TSS (правила доступа – такие же, как и для сегмента данных).

Уровни привилегий проверяются, когда селектор дескриптора сегмента загружается в сегментный регистр. Проверки, используемые при доступе к сегментам данным, отличаются от проверок, используемых при передаче управления другим сегментам кода. Ниже, в разделе 2.5.6, эти два типа проверок будут рассматриваться отдельно.

Уровень привилегий стека должен быть таким же, как и у кода, т.е. RPL селектора сегмента стека должен быть равен его DPL и CPL сегмента кода. В противном случае, генерируется исключение общей защиты.

2.5.5. Проверка уровня привилегий при доступе к сегментам данных

Для доступа к операнду в сегменте данных необходимо загрузить в сегментный регистр (DS, ES, FS или GS) селектор дескриптора сегмента данных. Для этого предназначены команды MOV, POP, LDS, LES, LFS и LGS.

Перед тем как загрузить селектор в сегментный регистр, процессор проверяет уровень привилегий, сравнивая уровень привилегий текущего кода (CPL), RPL селектора и DPL дескриптора. Процессор загружает селектор сегмента в регистр сегмента, если DPL в дескрипторе численно больше или равен и CPL и RPL. В противном случае загрузка не произойдет и процессор сгенерирует исключение общей защиты.

Уровень привилегий стека должен быть таким же, как и у кода, т. е. RPL селектора сегмента стека должен быть равен его DPL и CPL сегмента кода. В противном случае генерируется исключение общей защиты.

2.5.6. Проверка уровней привилегий при межсегментной передаче управления

Для передачи управления из одного сегмента кода в другой селектор целевого сегмента должен быть загружен в регистр CS. При загрузке процессор проверит в дескрипторе этого сегмента его предел, тип и уровень привилегий. Если проверка была успешной, то селектор будет загружен и произойдёт передача управления.

Передача управления осуществляется командами JMP, CALL, RET, SYSENTER, SYSEXIT, INT n и IRET.

Команда JMP или CALL может ссылаться на другой сегмент любым из четырёх способов:

- операнд команды содержит селектор целевого сегмента;
- операнд указывает на дескриптор шлюза вызова, который содержит селектор целевого сегмента;
- операнд указывает на TSS, в котором содержится целевой сегмент кода;
- операнд указывает на шлюз задачи, который указывает на TSS, где содержится целевой сегмент кода.

Далее будут описаны только два случая передачи управления, т. к. случаи передачи управления другим задачам описывались в разделе 2.4. Команды SYSENTER и SYSEXIT предназначены для быстрого вызова и возврата системных процедур; их действия будут рассмотрены ниже в этом разделе.

Близкие формы команд JMP, CALL и RET передают управление внутри текущего сегмента кода и проверка уровня привилегий не производится. Дальние формы этих команд передают управление в другой сегмент, и при этом происходит проверка уровня привилегий.

При передаче управления в другой сегмент кода без перехода через шлюз вызова процессор проверяет следующие значения:

- CPL сегмента кода, из которого происходит передача управления;
- DPL дескриптора сегмента кода, в который происходит передача управления.
- RPL селектора целевого сегмента;
- флаг согласованности в дескрипторе целевого сегмента кода (C=1/0 – сегмент согласованный/несогласованный).

Правила, по которым процессор проверяет CPL, RPL и DPL, зависят от значения флага C.

2.5.6.1. Передача управления несогласованному сегменту кода

При доступе к несогласованному сегменту кода CPL вызывающего кода должен быть равен DPL целевого сегмента кода, иначе процессор сгенерирует исключение общей защиты. RPL селектора несогласованного сегмента кода должен быть численно не больше CPL текущего кода. Когда такой селектор загружается в CS, поле привилегий CPL не меняется, даже если RPL имеет другое значение.

2.5.6.2. Передача управления согласованному сегменту кода

При передаче согласованным сегментам кода текущий уровень привилегий может быть численно равен или больше чем (т. е. менее привилегирован) DPL сегмента кода адресата; процессор генерирует исключение общей защиты, только если текущий уровень привилегий меньше, чем DPL. Также если управление передаётся согласованному сегменту кода, то RPL не проверяется.

Для согласованного сегмента кода DPL показывает численно наименьший уровень привилегий, который может иметь вызывающий код для передачи управления в него.

При передаче управления согласованному сегменту кода CPL не меняется, даже если DPL целевого сегмента меньше, чем CPL. Это единственная ситуация, когда CPL отличается от DPL. Кроме того, поскольку CPL не меняется, то смены стека не происходит.

Согласованные сегменты коды используются для модулей кода, таких как математические библиотеки и обработчики особых ситуаций, которые поддерживают приложения, но не требуют доступа к защищённым системным средствам. Эти модули являются частью операционной системы или исполнительной её части, но они могут быть выполнены на численно больших уровнях привилегий (менее привилегированных уровнях). То, что текущий уровень привилегий остаётся на уровне вызывающего кода при переходе к согласованному сегменту кода, препятствует прикладной программе получать доступ к более привилегированным ресурсам (коду и данным).

Обычно большая часть сегментов кода – не подчинённые, и в них передавать управление можно только на их уровне привилегий, кроме случаев перехода через шлюзы вызова, описанного ниже.

2.5.7. Шлюзы вызова

Для обеспечения контролируемого доступа к сегментам кода на различных уровнях привилегий процессор предоставляет четыре специальных типа дескрипторов, так называемых шлюзов:

1. Шлюз вызова
2. Шлюз ловушки
3. Шлюз прерывания
4. Шлюз задачи

Шлюзы ловушки и прерывания используются в дескрипторной таблице прерываний (IDT) и описаны в разделе 2.2 когда речь шла о прерывания в защищённом режиме. Шлюз задачи используется для передачи управления задаче и

подробно описывался в разделе 2.4, когда речь шла о многозадачности. Здесь будет описываться только шлюз вызова.

Шлюзы вызова обеспечивают передачу управления между различными уровнями привилегий. Также шлюз вызова полезен при передаче управления между 16-рязрядным и 32-разрядным кодом. Дескриптор шлюза вызова не может размещаться в IDT – он размещается в GDT или LDT; на рис. 2.30 приведён его формат.

Шлюз вызова выполняет шесть функций:

1. Определяет сегмент кода, к которому будет произведён доступ.
2. Определяет точку входа в этом сегменте.
3. Определяет уровень привилегий, который должна иметь вызывающая процедура.
4. При смене стека указывает число необязательных параметров, копируемых процессором из одного стека в другой.
5. Определяет размер значений, записываемых в целевой стек: 16-разрядный шлюз использует 16-разрядные значения, 32-разрядный – 32-разрядные.

Поле селектора сегмента в шлюзе вызова определяет сегмент кода, в которой будет находиться обработчик. Поле смещения определяет смещение точки входа в сегменте кода. Поле DPL содержит уровень привилегий шлюза вызова, в свою очередь являющийся уровнем привилегий, который должен иметь код, чтобы вызвать этот шлюз.

Поле количества параметров указывает количество параметров (16-битных или 32-битных – в зависимости от типа сегмента кода) для копирования из стека вызывающей процедуры в стек вызываемой, если происходит переключение стека.

Обычно флаг P всегда установлен; если же он сброшен, то при обращении к шлюзу процессор генерирует исключение отсутствующего сегмента (`#NP`). Используя бит P, система может определить число переходов через этот шлюз. Для этого бит присутствия шлюза сбрасывается, а код функции обработчика исключения `#NP` сводится к увеличению на 1 счётчика переходов через шлюз и установке бита P. После возврата из такого обработчика процессор снова попытается передать управление через шлюз и сделает это, если больше никаких нарушений в нём не обнаружит. Такой способ не очень эффективен, но позволяет определить число переходов через шлюз даже в том случае, если функция-обработчик шлюза будет изменена другими программами, работающими на нулевом уровне привилегий.

Чтобы обратиться к шлюзу вызова, необходимо выполнить дальний вызов или прыжок командами CALL или JMP, указав при этом селектор необходимого шлюза

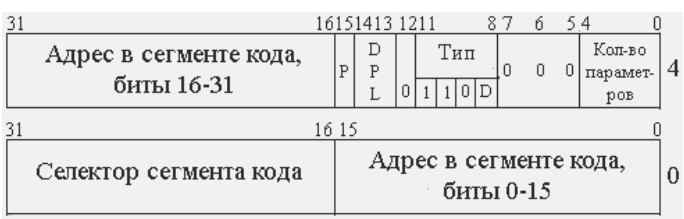


Рис. 2.30. Формат дескриптора шлюза вызова

вызова. Селектор в операнде должен иметь значение селектора требуемого шлюза; смещение при вызове шлюза вызова игнорируется.

При передаче управления через шлюз вызова сравниваются четыре значения уровней привилегий: CPL, RPL, DPL дескриптора шлюза вызова, DPL дескриптора целевого сегмента кода. Также проверяется флаг согласованности. Правила проверки уровня привилегий отличаются в зависимости от того, какой командой было передано управление.

1. В обоих случаях передачи управления CPL и RPL не могут быть больше, чем DPL шлюза вызова.
2. Также в обоих случаях передачи управления DPL целевого сегмента кода должен быть не больше, чем CPL, если целевой сегмент кода согласованный.
3. Если целевой сегмент кода несогласованный, то проверки различаются:
   a. при передаче управления командой CALL – DPL целевого несогласованного сегмента должен быть не больше, чем CPL.
   b. при передаче управления командой JMP – DPL целевого несогласованного сегмента кода должен быть равен CPL.

Поле DPL шлюза вызова определяет численно самый высокий уровень привилегий, от которого вызывающий может обратиться к шлюзу вызова; т. е., чтобы обращаться к шлюзу вызова, CPL текущего кода должен быть равен или меньше DPL шлюза вызова. RPL селектора шлюза вызова подвергается тем же проверкам, что и CPL; т. е. RPL должен быть меньше или равен DPL шлюза вызова.

После проверки привилегий между процедурой запроса и шлюзом вызова процессор приступает к проверке DPL целевого сегмента кода с CPL. Здесь правила проверки привилегии изменяются между командами CALL и JMP. Только команда CALL может передать управление более привилегированному несогласованному сегменту кода, т. е. к несогласованному сегменту кода, DPL которого меньше, чем CPL. Команда JMP может передать управление только к несогласованному сегменту кода, DPL которого равен текущему уровню привилегий. Обе команды могут передать управление более привилегированному согласованному сегменту кода, т. е. согласованному сегменту, DPL которого меньше или равен CPL.

Если запрос сделан к более привилегированному (численно более низкий уровень привилегий) несогласованному сегменту кода, то текущий уровень привилегий понижается до DPL целевого сегмента кода и происходит переключение стека. Если вызов сделан к более привилегированному согласованному сегменту кода, то текущий уровень привилегий не изменяется и переключения стека не происходит.

Шлюзы вызова позволяют коду, расположенному на менее привилегированном уровне привилегий, вызывать процедуры и функции, расположенные на более привилегированных уровнях. Операционная система может запретить прикладным программам обращение к устройствам напрямую и предоставить свои сервисы для работы с устройствами через шлюзы вызова. Такой метод разграничения доступа программ к устройствам используется в большинстве операционных систем.

2.5.8. Переключение стека

Всякий раз, когда шлюз вызова используется, чтобы передать управление процедуре на более привилегированном несогласованном сегменте кода (т. е. когда DPL несогласованного сегмента кода меньше, чем текущий уровень привилегий), процессор автоматически переключается на стек целевого сегмента кода. Переключение стека выполняется, чтобы более привилегированный код не мог завершиться с ошибкой из-за того, что места в стеке недостаточно. Это также защищает более привилегированный код от вмешательства менее привилегированного кода через стек.

Каждая задача может иметь до четырех стеков: для третьего уровня и для каждого из используемых привилегированных уровней. Указатели на стеки нулевого, первого и второго уровней привилегий хранятся в TSS. Каждый из этих указателей состоит из селектора сегмента и указателя на вершину стека. Процессор не изменяет их, в то время пока задача выполняется. Они используются только, чтобы создать новый стек, когда выполняется вызов кода на более привилегированный уровень. При возвращении к вызвавшему коду стек больше не используется и обычно он освобождается. В следующий раз, когда произойдёт вызов более привилегированного кода, новый стек вновь создаётся, используя указатели вершины стека в TSS. Значения, описанные в полях для SS:ESP в TSS, обычно соответствуют указателю на стек третьего уровня привилегий. Для стека третьего уровня привилегий не выделено отдельного поля в TSS, потому что процессор не допустит передачи управления с нулевого, первого и второго уровней на третий уровень, за исключением возврата и привилегированной процедуры.

За создание и определение стеков должна отвечать сама операционная система, и в каждом стеке должно хватать места для хранения следующих значений:

1. Содержимое регистров вызывающей процедуры: SS, ESP, CS и EIP.
2. Параметров, передаваемых вызываемой процедуре и временных переменных, используемых ей.
3. Регистра EFLAGS и кода ошибки (в случае вызова обработчика исключения или прерывания).

Если операционная система не использует механизм многозадачности, то она всё равно должна определить как минимум одну TSS для переключения стеков.

Когда вызов процедуры через шлюз вызова приводит к изменению уровня привилегий, процессор исполняет следующие действия:

1. Используя значение DPL целевого сегмента кода (оно и будет новым значением CPL), выбирает из TSS указатель на новый стек.
2. Осуществляет проверку селектора и смещения нового стека; если при этом были обнаружены нарушения, генерируются исключения недопустимого TSS (`#TS`).
3. Проверяет дескриптор сегмента стека на соответствующие привилегии и его тип и генерирует исключение недопустимой TSS, если при этом будут обнаружены нарушения.
4. Временно сохраняет текущие значения SS и ESP.
5. Загружает новые значения регистров SS и ESP.

6. Помещает в новый стек временно сохранённые значения регистров SS и ESP.
7. Копирует указанное в дескрипторе шлюза число параметров из стека вызывающей процедуры в новый стек. Если счётчик параметров равен 0, то параметры не копируются.
8. Помещает в новый стек адрес возврата в вызвавшую процедуру (т. е. текущее значение регистров CS и EIP).
9. Загружает новые значения в регистры CS и EIP из шлюза вызова и начинает выполнение вызванной процедуры.

Самое время вспомнить инструкцию RETF. Для того чтобы вернуться из обработчика шлюза вызова, надо выполнить инструкцию RETF. Инструкция RETF выполняет дальний возврат управления, т. е. извлекает из стека значение регистра CS и EIP и, если целевой уровень привилегий отличается от текущего, то происходит переключение стека.

Возвращение управления командой RETF со сменой уровня привилегий возможно лишь только на менее привилегированный уровень привилегий (т. е. DPL целевого сегмента кода численно больше, чем CPL). Процессор использует поле RPL регистра CS, который был сохранён в стеке вызванной процедуры, чтобы определить, требуется ли возвращение к численно более высокому уровню привилегий. Если RPL численно больше (менее привилегированный) чем текущий уровень привилегий, то происходит возвращение на другой уровень привилегий; в противном случае генерируется исключение общей защиты. После вышеуказанной проверки производятся действия, обратные тем, которые производились при передаче управления.

2.5.9. Использование инструкций SYSENTER и SYSEXIT

Для быстрого вызова системных привилегированных процедур в процессорах были введены инструкции SYSENTER и SYSEXIT. Команда SYSENTER используется для вызова системной процедуры и может быть вызвана с любого уровня привилегий. Команда SYSEXIT предназначена для возврата из системной процедуры и может быть вызвана только на нулевом уровне привилегий; возврат всегда происходит на третий уровень привилегий. Данные инструкции оптимизированы для максимальной производительности при передаче управления и третьего уровня привилегий на нулевой. Использовать этот механизм в других случаях не имеет смысла.

Команды SYSENTER и SYSEXIT – сопутствующие команды, но они не составляют пару командам CALL/RET. Также команда SYSENTER не сохраняет информации о состоянии для команды SYSEXIT. Данные команды не имеют операндов, операнды жёстко задаются в регистрах MSR.

Для команды SYSENTER значения извлекаются из следующих источников:

1. Из регистра IA32_SYSENTER_CS извлекается селектор целевого сегмента кода.
2. Точка входа задаётся в регистре IA32_SYSENTER_EIP.

3. Селектор целевого сегмента стека вычисляется путём добавления 8 к значению из IA32_SYSENTER_CS.
4. Указатель на стек задаётся в регистре IA32_SYSENTER_ESP.

Для команды SYSEXIT:

1. Селектор целевого сегмента кода вычисляется путём добавления 16 к значению из IA32_SYSENTER_CS.
2. Точка входа считывается из регистра EDX.
3. Селектор целевого сегмента стека вычисляется путём добавления 24 к значению из IA32_SYSENTER_CS.
4. Указатель на стек считывается из регистра ECX.

Как видно, команды SYSENTER/SYSEXIT производят минимум обращений к памяти, а следовательно, предоставляют максимально быстрый способ вызова системныхй процедур. Для осуществления механизма вызова системных процедур через команды SYSENTER/SYSEXIT нам потребуется особым образом разместить дескрипторы сегментов кода и стека в GDT и задать соответствующие значения в трёх регистрах MSR.

Регистры IA32_SYSENTER_CS, IA32_SYSENTER_EIP и IA32_SYSENTER_ESP на всех процессорах имеют одинаковые индексы (табл. 2.5).

Таблица 2.5. Индексы MSR-регистров для команд SYSENTER/SYSEXIT

IA32_SYSENTER_CS	174h
IA32_SYSENTER_ESP	175h
IA32_SYSENTER_EIP	176h

2.5.10. Практика

Напоследок не помешало бы немного практики. Наиболее целесообразно будет написать небольшой пример с использованием инструкций SYSENTER/SYSEXIT.

В следующем примере мы напишем программу защищённого режима, которая будет разделена на две части: код третьего кольца и код нулевого кольца. Ничего особенного они не делают – всего лишь выводят сообщение. Код третьего кольца будет находиться в бесконечном цикле, для вывода сообщения будет вызывать код нулевого кольца через команду SYSENTER. Код нулевого кольца будет только выводить сообщение и возвращать управление.

Основные константы и структуры для максимального удобства вынесены в файл pmstructures.asm. Теперь пойдем по порядку: пункт первый – содержимое файла PM_CODE.ASM.

Листинг 2.30. Обновление GDT

```
include 'pmstructures.asm';

Ring0Stack     equ  200000h  ; стек нулевого кольца
Ring3Stack     equ  180000h  ; стек третьего кольца
```

```
CS_r0  equ 8h  ; селектор для кода нулевого кольца
DS_r0  equ 10h ; селектор для стека третьего кольца

 START_CODE:

   mov esi, message1
   mov al, 0
   mov ah, 0
   mov bl, "5"
   call OutText  ; вызов процедуры вывода строки на экран

   sgdt [GDTR_Image]
   mov eax, NewGDT
   mov [GDTR_Image.BaseAddress], eax
   mov ax, NewGDTEnd - NewGDT - 1
   mov [GDTR_Image.Limit], ax

   lgdt [GDTR_Image]

   jmp  CS_r0:@f
  @@:
   mov ax, DS_r0
   mov ds, ax
   mov ss, ax
```

В начале выполнения программы мы выводим сообщение о том, что процессор работает в защищённом режиме. Потом программа обновляет глобальную дескрипторную таблицу, т. к. нам надо добавить два дескриптора для третьего кольца и расположить все дескрипторы в особом порядке. Для этого мы просто загружаем в регистр GDTR указатель на новую таблицу. После чего обновляем сегментные регистры «на всякий пожарный», чтобы убедиться, что у нас всё правильно получилось.

Листинг 2.31. Инициализация механизма SYSENTER/SYSEXIT

```
   mov esi, message2
   mov al, 0
   mov ah, 1
   mov bl, "5"
   call OutText

   xor edx, edx

   mov ecx, IA32_SYSENTER_CS
   mov eax, CS_r0
   wrmsr

   mov ecx, IA32_SYSENTER_CS
   mov eax, CS_r0
   wrmsr
```

```
    mov ecx, IA32_SYSENTER_EIP
    mov eax, Ring0Code
    wrmsr

    mov ecx, IA32_SYSENTER_ESP
    mov eax, Ring0Stack
    wrmsr

    mov esi, message3
    mov al, 0
    mov ah, 2
    mov bl, "5"
    call OutText
```

В листинге 2.31 мы выводим сообщение о том, что обновлена глобальная дескрипторная таблица. После чего инициализируем регистры MSR и выводим соответствующее сообщение.

Листинг 2.32. Код третьего и нулевого кольца, использующий механизм SYSENTER/SYSEXIT

```
    mov edx, Ring3Code
    mov ecx, Ring3Stack
    sysexit

Ring3Code:
    mov ax, DS_r3
    mov ds, ax

    mov esi, message4
    mov al, 0
    mov ah, 3
    mov bl, "5"
    call OutText

   @@:
    inc [counter]
    mov edx, esp
    sysenter

   sysenter_ret:
    jmp @b

Ring0Code:
    mov ax, DS_r0
    mov ds, ax

    mov esi, message5
    mov al, 0
    mov ah, 4
```

```
mov bl, "5"
call OutText

mov ecx, edx
mov edx,sysenter_ret
sysexit
```

После вывода сообщения о том, что программа инициализировала структуры для команд SYSENTER/SYSEXIT, мы вызываем команду SYSEXIT для выхода в третье кольцо. В коде третьего кольца мы выводим соответствующее сообщение и в бесконечном цикле выводим сообщение с переходом в нулевое кольцо, увеличивая при этом счётчик на единицу. Поскольку значение счётчика мы не преобразуем в строку, то после текста сообщения должны мелькать «закорючки».

В данном примере код третьего кольца почти ничем не ограничивается и может сам выводить сообщение. Вывод сообщения в нулевом кольце предусмотрен только лишь для демонстрации работы инструкций SYSENTER/SYSEXIT. При программировании реальной операционной системы, разумеется, следует позаботиться о перезагрузке сегментных регистров ES, FS, GS, т. к. сегменты данных, используемые кодом третьего и нулевого кольца, могут отличаться.

Приводить остальной код не имеет смысла – полный исходный код примера находится в папке part2 на компакт-диске, прилагающемся к данной книге.

2.5.11. Резюме

В этом разделе было завершено описание защищённого режима процессора. Нами были изучены основополагающие механизмы, обеспечивающие защиту данных операционной системы от обычных программ. Ниже пойдёт речь о программировании под 32- и 64-битными операционными системами Windows, а также будет приведено описание работы процессора в 64-битном режиме.

3 ПРОГРАММИРОВАНИЕ В WIN32

3.1. Введение в Win32

В предыдущей главе мы говорили о защищённом режиме процессора, а точнее, об основах построения любой операционной системы: механизмах защиты, многозадачности, управления памятью, обработки прерываний. Тем не менее создание программ, которые могут работать без операционной системы, – очень сложная, можно даже сказать, инженерная задача. Создание такой программы, по сути, всегда сводится к созданию мини-ОС.

В данной главе мы будем говорить о программировании на ассемблере под операционной системой Windows, точнее под 32-битной её версией.

Итак, что же такое Windows или Win32? Win32 – это семейство многозадачных операционных систем защищённого режима.

Первая 32-битная операционная система Windows вышла в 1995 году под названием Windows 95. Её версия имела номер 4.0 (все предыдущие версии не были полноценными операционными системами, а лишь предоставляли интерфейс к возможностям операционной системы MS-DOS). В 1998 году вышла Windows 98 (Windows 4.10), в 2000 году – Windows Millennium Edition (Windows 4.90). Все эти системы относятся к семейству Windows 9x. Основной стратегической задачей создания семейств Windows 9x являлся перевод пользователей на новые 32-битные программы при сохранении преемственности программ, написанных для MS-DOS. Поскольку MS-DOS, входившая в состав данных программных продуктов, предоставляла полный доступ ко всем периферийным устройствам, данным и коду операционной системы, данным и коду драйверов устройств, а также данным других программ, исполняющихся в системе, семейство программных продуктов Windows 9x допускало умышленную или неумышленную порчу содержимого оперативной памяти, что могло послужить одной из причин «зависания» или некорректной работы системы.

Одновременно с разработкой систем Windows 9x велась разработка новой линейки полностью 32-битных систем, не нуждавшихся в поддержке со стороны MS-DOS, как это было в предыдущих версиях и в Windows 9x. Эта линейка систем получила название Windows NT. Первая операционная система этой линейки вышла в 1993 году под названием Windows NT 3.1. Потом выходили версии Windows NT 3.5 (1994), Windows NT 3.51 (1995), Windows NT 4.0 (1996).

Настоящим прорывом в линейке NT стал выпуск системы Windows 2000 (2000 год). Потом увидела свет операционная система Windows XP (Windows NT 5.1, 2001 год). Ну а что было дальше, всем известно. Одна за другой появлялись

серверная система Windows 2003 Server, 64-битная версия Windows XP, Windows Vista и т. д.

Поскольку семейство операционных систем Win9x является семейством неполноценных 32-битных систем и на момент написания данной книги они являются устаревшими, то говорить об аспектах программирования в этих операционных системах мы не будем. Наиболее подробно и конкретно обсудим программирование под операционными системами семейства NT.

3.1.1. Основные сведения

Как уже было сказано, системы WinNT – это операционные системы защищённого режима (вернее, их 32-битные версии). Система безопасности этих операционных систем построена на разделении кода пользователя и системного кода. Не следует воспринимать словосочетание «код пользователя» буквально: под кодом пользователя подразумевается код любого приложения, работающий в третьем кольце. На него наложено очень много ограничений. Системный код работает в нулевом кольце и почти ничем не ограничен. Режим, в котором код работает в третьем кольце, называется user mode; режим, в котором код работает на нулевом уровне привилегий, называется kernel mode.

Основанная единица выполнения в Win32 – это *поток* (по терминологии защищённого режима – задача). Потоки объединяются в *процессы*. В общем случае одна программа – это один процесс. В процессе может быть сколько угодно потоков. Каждый процесс обособлен от всех остальных. Это достигается за счет того, что у каждого процесса своя собственная виртуальная память. Тем не менее, если программе надо получить доступ в памяти других процессов, она может осуществить это через специальные системные сервисы.

Диспетчеризация потоков осуществляется на основе приоритетов. Приоритеты у потоков не являются статическими параметрами – они могут меняться в зависимости от того, что делает поток. Так или иначе, в некоторый момент времени на центральном процессоре всегда выполняется поток с наибольшим приоритетом (впрочем, это довольно-таки сложная тема, и в двух словах её нельзя описать; она заслуживает отдельной главы). В третьем кольце выполняется большая часть исполняемых компонентов в Windows. Также следует указать, что многозадачность во всех системах Windows реализуется не аппаратно, а программно.

3.1.2. Память в Win32

Как уже было сказано выше, каждый процесс имеет своё собственное виртуальное адресное пространство. Адресное пространство любого процесса разбито на две равные части: память процесса и память системы. Младшие 2 Гб памяти являются памятью процесса, старшие 2 Гб – памятью системы. Память системы одна для всех процессов, она недоступна из user mode даже для чтения; любое обращение к этой памяти приводит к ошибке доступа к памяти и завершению приложения.

В некоторых случаях память системы и процесса делятся не поровну: под память процесса выделяется 3 Гб памяти, а под память системы – 1 Гб. Так делается в тех случаях, когда используются приложения, требовательные к памяти, которым

2 Гб памяти недостаточно. После выхода 64-битных систем такой метод разделения памяти потерял свою актуальность.

Адреса в диапазоне 0h–FFFFh никому не доступны – эта память нужна для выявления нулевых указателей. Любой указатель, значение которого меньше 10000h, считается нулевым. Таким образом, каждому процессу в Win32 в общем случае доступно 2 Гб (за вычетом 64 Кб) виртуальной памяти.

3.1.3. Исполняемые компоненты Windows

В операционных системах Windows имеется несколько типов исполняемых файлов. Все типы исполняемых файлов имеют формат PE (Portable Executable). Наиболее часто используемые исполняемые компоненты в Windows: EXE (приложение), DLL (динамическая библиотека), SYS (драйвер). EXE-файлы – это самый распространённый тип исполняемых файлов в Windows, в них находятся наши программы. DLL-файлы – это динамически загружаемые библиотеки, в них хранятся функции и процедуры, которые могут использовать другие исполняемые компоненты. SYS-файлы – это файлы драйверов режима ядра, в них находится код нулевого кольца операционной системы.

Файлы формата PE состоят из заголовка и секций. Секция в PE-файле – это его основная составляющая единица. В заголовке содержатся основные характеристики файла и таблица секций. Описание формата PE выходит за рамки этого раздела и требует отдельного внимания, но всё же для максимального усвоения дальнейшего материала надо получить о формате исполняемых файлов базовые сведения. Самые главные характеристики файла – точка входа и база образа, т. е. указание, по какому адресу должен быть загружен данный модуль. Линкер при создании исполняемого файла должен вставить в код программы вместо меток некоторые конкретные адреса и подразумевает, что этот код будет загружен по некоторому базовому адресу. Загрузчик Windows должен знать, по какому адресу надо загрузить данный исполняемый файл, и именно для этого используется поле базы образа в заголовке исполняемого файла. Очень часто при загрузке файлов DLL и SYS адрес, указанный в поле базы образа, является уже занятым или простонапросто недоступным. Если при загрузке исполняемого файла адрес, указанный в поле базы образа, уже занят, то он грузит файл по другому адресу, и при этом загрузчику надо подправить в коде программы все обращения к данным. Для этого загрузчику будут нужны релокейшены. *Релокейшены* содержат информацию о командах, в которых есть обращения к памяти, для поправки адресов. Точка входа содержит адрес, с которого начнётся выполнение исполняемого файла.

Каждый файл может иметь таблицу импорта и экспорта. С помощью таблицы импорта исполняемый файл может импортировать функции, которые находятся в других модулях, загруженных в текущее адресное пространство, и использовать эти функции как свои. Для того чтобы другие модули могли использовать функции из данного модуля, адреса этих функций должны быть прописаны в таблице экспорта.

Как было сказано выше, в заголовке PE-файла содержится таблица секций, она описывает каждую секцию в PE-файле: начало данных в секции, размер данных,

адрес, куда должна быть спроецирована данная секция, и её характеристики. Наиболее часто в секциях находятся данные, код, импорты, экспорты и релокейшены.

Файлы с расширением EXE являются обычными программами; в 99,99% случаев EXE-файл представляет процесс, в память которого он загружен. Файлы с расширением DLL являются библиотеками, где содержатся функции, которые могут использовать другие программы или другие DLL. Файлы с расширением SYS являются драйверами режима ядра; код, содержащийся в них, выполняется на нулевом уровне привилегий, в режиме ядра. В файлах DLL и SYS точка входа указывает на инициализирующую функцию.

3.1.4. Системные библиотеки и подсистемы

В коде Win32, выполняемом в режиме пользователя (в третьем кольце), запрещены любые прямые обращения к устройствам и портам ввода-вывода. Что это значит? Это значит, что любые обращения к портам ввода/вывода, вызов прерываний и выполнение привилегированных инструкций приведут к ошибке и завершению программы; обращение к памяти, на которую спроецированы регистры устройств, обращение к другим важным областям памяти (например, 0B8000h) ничего не даст. Без обращения к внешним устройствам и портам ввода-вывода польза от программ, работающих в третьем кольце, нулевая. Для того чтобы они могли обратиться к внешним устройствам и наладить взаимодействие с «внешним миром», операционная система предоставляет программам API-функции. Все API-функции содержатся в системных DLL-библиотеках, самые главные из них: kernel32.dll (взаимодействие с системой), user32.dll (пользовательский интерфейс), gdi32.dll (графика). Функции библиотеки kernel32.dll в основном являются оболочками вокруг функций из ntdll.dll.

Функции из библиотеки ntdll.dll являются «переходниками» к функциям ядра Windows: эти функции просто принимают параметры, подготавливают их к вызову шлюза ядра (Win2000) или вызову команды SYSENTER (WinXP и позже) и вызывают сервис ядра через шлюз int 2Eh или команду SYSENTER. Библиотека ntdll.dll – важнейший компонент пользовательской подсистемы Windows.

Во время разработки систем WinNT подразумевалось, что они должны будут поддерживать исполняемые файлы других операционных систем, например MS-DOS, POSIX, OS/2 и пр. Для каждой подсистемы, кроме MS-DOS, в операционной системе есть свои библиотеки. Программы, предназначенные для MS-DOS, выполняются на виртуальной машине NTVDM – вот почему старые программы для DOS не вызывают никаких ошибок при выполнении. Также в Windows, как ни странно, есть и подсистема Win32.

Библиотека ntdll.dll является основополагающей для всех подсистем, т. к. именно через неё пользовательский код может взаимодействовать с кодом ядра. Эта библиотека загружается в память любого процесса одной из первых и всегда по одному и тоже адресу. Библиотека kernel32.dll является основополагающей подсистемы Win32 – она загружается во все процессы Win32 одной из первых (разумеется, после ntdll) и всегда по одному и тому же адресу.

3.1.5. Модель вызова функций в Win32

В системах Win32 при вызове всех системных функций используется модель вызова stdcall. Согласно этой модели параметры функций передаются через стек в обратном порядке, при этом за очистку стека от параметров ответственна вызываемая функция. Например, если у функции есть три параметра, то вызов согласно модели stdcall будет выглядеть так:

```
Push param3
Push param2
Push param1
Call FunctionAddr
```

Результат выполнения функции будет содержаться в регистре EAX.

Также при использовании API-функций следует помнить, что они сохраняют значение не всех регистров общего назначения. Соглашение stdcall предусматривает сохранение содержимое регистров EBX, ESI, EDI и EBP. Также при написании функций обратного вызова (подробнее о них пойдет речь в разделе 3.2) надо обязательно сохранять содержимое этих регистров, поскольку код системных функций не ожидает их изменения.

Пример создания функций, отвечающих соглашению вызова stdcall, будет приведён в параграфе 3.1.7.2.

3.1.6. Выполнение программ в Win32: общая картина

Итак, как же выглядит общая картина? Обрисуем её последовательно.

Когда загрузчик Windows загружает наш исполняемый файл, то сначала создаёт для него виртуальное адресное пространство размером 4 Гб, причём нижние 2 Гб из них доступны приложению. Потом он загружает системные библиотеки ntdll.dll и kernel32.dll; если в таблице импорта нашего файла указаны ещё какие-нибудь библиотеки, то они загружаются тоже. После того как библиотеки загружены, создаётся первичный поток процесса, который начинает своё выполнение с точки входа программы.

Во время выполнения процесса, вернее его потоков, ему запрещены какие-либо обращения к портам ввода/вывода и вызов каких-либо прерываний. Также запрещены работа с привилегированными регистрами и выполнение привилегированных команд. Из-за вышеуказанных ограничений работа программы лишена всякого смысла – невозможно даже вывести на экран сообщение. Чтобы программы могли работать с внешними устройствами «приносить пользу», Windows предоставляет им API-функции, позволяющие программам работать с внешними устройствами, взаимодействовать с системой, а также друг с другом. API-функции находятся в системных библиотеках; каждая функция, которая работает с ресурсом, охраняемым системой (файлы, процессы, устройства и т. д.), вызывает соответствующую функцию ядра системы.

Вызов подавляющего большинства функций, экспортируемых системными библиотеками функций, происходит по соглашению stdcall.

Резюмируем всё вышесказанное: операционная система помещает программу в некоторое изолированное адресное пространство, разрешая ей взаимодействовать

с «внешним миром» посредством функций (системных сервисов), которые сама же и предоставляет. Операционная система Windows избавляет программиста, который пишет программы на ассемблере, от множества забот, которые вообще-то не должны его касаться, например: от работы с системными регистрами и структурами, взаимодействия с внешними устройствами, реализации работы с файловой системой и т. д. В связи с этим программирование на ассемблере под Win32 намного легче, чем многие думают. При работе с файлами программисту не нужно заботиться о том, какая же модель жёсткого диска установлена на компьютере: достаточно просто вызывать функции, которые предоставляет операционная система, а она уже сама разберётся со всеми проблемами.

3.1.7. Практика

3.1.7.1. Первая программа под Win32

После того как мы получили базовые сведения о программировании в Win32, пора приступить к конкретике и написать программу, выводящую знаменитую фразу «Hello world!».

Текст программы на FASM, который выводит сообщение «Hello world!», приведён в листинге 3.1.

Листинг 3.1. Исходный код простейшей программы для Win32

```
format PE GUI 4.0

include 'win32a.inc'

entry start

section '.data' data readable writeable

caption db 'First Win32 program',0
Message db 'Hello World!',0

section '.code' code readable executable

start:
     push 0
     push caption
     push Message
     push 0
     call [MessageBox]

     push 0
     call [ExitProcess]

section '.relocs' fixups readable writeable

section '.idata' import data readable writeable
```

```
library kernel,'KERNEL32.DLL',\
   user,'USER32.DLL'

import kernel,\
   ExitProcess,'ExitProcess'

import user,\
   MessageBox,'MessageBoxA'
```

Начнём сначала. Первая строка `format PE GUI 4.0` говорит компилятору, что мы хотим получить на выходе исполняемый файл формата PE. Директива GUI 4.0 предписывает ему указать в характеристиках файла, что приложение будет оконное. Потом мы включаем файл win32a.inc с макросами, упрощающими создание приложения для Win32. Почему в конце буква A? В Win32 у всех функций, которые принимают строку (напрямую или косвенно), есть две версии: ANSI и UNICODE. Если в конце имени функции стоит буква A, то функция принимает ANSI-строки, а если буква W – то UNICODE-строки. Файл win32a содержит макросы и константы для ANSI-версий функций.

В директиве `include` необходимо указывать относительные или абсолютные пути. Постоянно указывать пути типа `..\..\include\win32a.inc` или `D:\Programs\assemblers\FASM\include\win32a.inc` довольно-таки неудобно. Для того чтобы можно было просто указать имя включаемого файла (win32a.inc), нужно прописать в файл настроек FASMW.INI абсолютный путь к папке INCLUDE:

```
[Environment]
include=E:\FASMW166\include
```

Возвращаемся к нашей программе. Директивой `entry` мы задаём точку входа программы, т. е. метку, с которой начнётся её выполнение.

Директивой `section` задаётся секция в PE-файле. Секции имеют следующие параметры: имя (без нуля в конце и максимум 8 символов), тип секции (код, данные, релокейшены, импорты, экспорты) и атрибуты доступа (чтение, запись, выполнение). Теоретически можно создать только одну секцию и поместить всё (код, данные, импорты) в неё, но правильнее будет создать для кода, данных и импортов отдельную секцию.

В секции данных нет ничего сложного – там объявлены две строки: заголовок окна сообщения и текст самого сообщения. Важно знать, что все строки должны оканчиваться нулевым байтом. Функции, принимающие строки, «узнают», где заканчивается строка, именно по нулевому байту в конце; если нулевого байта не будет, то будет выводиться длинная последовательность символов, пока он не встретится. Это общепринятое правило при передаче строк для подавляющего большинства функций. При использовании UNICODE-строк за конец строки будут приниматься два нулевых байта.

Итак, самое главное – секция кода. Для вывода сообщения в Win32 есть очень много способов и функций, но самая простая из них – это функция `MessageBox`. Функция `MessageBox` принимает четыре параметра. Первый параметр – это хендл окна владельца, т. е. окно, относительно которого наше окно сообщения будет

модальным: если 0, то окно сообщения не будет иметь владельца. Второй параметр – это сам текст сообщения, указатель на строку, заканчивающуюся нулём. Третий параметр – это заголовок нашего окна сообщения, указатель на строку, заканчивающуюся нулём. Последний, четвёртый параметр – это флаги, которые задают тип окна сообщения, т. е. какие кнопки будут на нём и т. д.: если 0, то будет только одна кнопка – OK. Функция возвращает код кнопки, на которую нажали; в нашем случае нам это не нужно. С учётом того что параметры надо передавать в обратном порядке, вызов функции будет выглядеть так:

```
push 0
push caption
push Message
push 0
call [MessageBox]
```

После вызова функции нам надо завершить выполнение программы. Для завершения работы программы по идее достаточно просто вернуть управление командой `ret`, но мы воспользуемся функцией `ExitProcess`. Функция принимает один параметр – код выхода процесса, чтобы в другом приложении модно было понять, почему процесс завершился. В данном случае нам это не надо. Таким образом, вызов функции `ExitProcess` и завершение программы осуществляются так:

```
push 0
call [ExitProcess]
```

После секции с кодом программы идёт секция релокейшенов (атрибут `fixups`); для EXE-файлов эта секция в 99,99% случаев просто не нужна, но мы всё равно её создадим. Последняя секция – это секция с таблицей импорта. Программа использует таблицу импорта для получения адресов функций, экспортируемых системными библиотеками. При загрузке программы в память загрузчик анализирует таблицу импорта, получает список функций, необходимых программе, и список модулей, из которых она хочет импортировать функции, после чего помещает в таблицу импорта адреса функций, имена которых в ней указаны.

Макросы library и import максимально упрощают создание таблицы импорта программы. Сначала мы указываем, в каких библиотеках находятся импортируемые нами функции, затем – из какой библиотеки какие функции импортируются. Описание самого формата таблицы импорта выходит за рамки данного раздела, и здесь он рассматриваться не будет.

Следует обратить внимание на объявление функции `MessageBox`; в её имени мы указали `MessageBoxA`. Это значит, что из библиотеки USER32.DLL мы импортируем функцию `MessageBoxA`, т. е. ту версию, которая работает с ANSI-строками.

3.1.7.2. Функция STDCALL

В качестве второго примера рассмотрим пример написания функции, отвечающей соглашению вызова stdcall.

Прежде чем приступить к написанию программы, необходимо изучить несколько основных понятий, которые пригодятся нам при написании любой функции. У каждой функции и подпрограммы есть участок стека, принадлежащий только ей; в

этом участке стека хранятся параметры, переданные ей, локальные переменные и адрес возврата. Этот участок стека называется *стековым фреймом функции*.

Указателем на стековый фрейм служит регистр EBP. Отчасти из-за того, что так сложилось исторически, отчасти из-за того, что для создания и удаления стекового фрейма предназначены команды ENTER/LEAVE, эти команды используют EBP как указатель стекового фрейма.

Итак, начнём сначала: что делает инструкция ENTER? Она принимает два параметра: первый указывает размер локальных данных, второй задаёт вложенность стекового фрейма. Команда ENTER сохраняет регистр EBP в стеке, после чего присваивает ему значение регистра ESP, а потом вычитает из ESP число, равное размеру локальных данных. Если указать вторым параметром 0, то стековый фрейм будет рассматриваться как не вложенный. Вложенные стековые фреймы рассматриваться не будут из-за отсутствия их практического применения.

Команда LEAVE не принимает параметров и производит две операции: она присваивает регистру ESP значение EBP, после чего восстанавливает значение EBP вызвавшей функции из стека. Сохранение регистра ESP в EBP даёт нам возможность любых манипуляций со стеком – в любом случае команда LEAVE восстановит предыдущее начальное значение ESP, которое хранится в регистре EBP.

Параметры функции имеют положительное смещение относительно регистра EBP, а локальные переменные – отрицательное смещение. На рис. 3.1 приведён пример стекового фрейма функции.

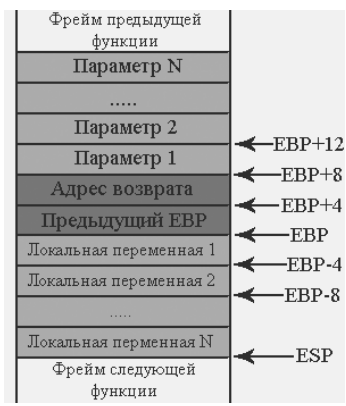


Рис. 3.1. Стековый фрейм функции

В листинге 3.2 приведён код функции, которая принимает четыре параметра и имеет две локальные переменные, каждая размером 4 байта.

Листинг 3.2. Пример stdcall-функции

```
Func:
   enter 8, 0
   mov eax, [ebp+8] ; eax = первый параметр функции
   mov [ebp-4], eax ; первая локальная переменная = eax
   mov ebx, [ebp+16]; ebx = третий параметр функции
```

```
   mov [ebp-8], ebx ; вторая локальная переменная = ebx
...
...
   leave
   retn 16
```

На практике инструкция ENTER почти не применяется – вместо неё используются инструкции `push ebp ; mov ebp, esp`. Обычно функции не используют локальные переменные и вложенные стеки; таким образом, если бы инструкция ENTER применялась, то с обоими нулевыми параметрами. Опкод инструкции всегда занимает 4 байта независимо от операндов, а пара инструкций PUSH/MOV в сумме занимает 3 байта. Даже если функция будет использовать локальные переменные, то инструкции PUSH/MOV/SUB ESP будут выполняться в пять раз быстрее, чем одна команда ENTER. Ситуация с инструкцией LEAVE немного другая: на практике чаще используется LEAVE, нежели пара `mov esp, ebp; pop ebp`.

Далее в листинге 3.3 приведена функция без использования инструкции ENTER.

Листинг 3.3. Пример stdcall-функции без инструкции ENTER

```
Func:
   push ebp
mov ebp, esp
sub esp, 8
   mov eax, [ebp+8] ; eax = первый параметр функции
   mov [ebp-4], eax ; первая локальная переменная = eax
   mov ebx, [ebp+16]; ebx = третий параметр функции
   mov [ebp-8], ebx ; вторая локальная переменная = ebx
...
...
   leave
   retn 16
```

А теперь перейдем к практике. Напишем программу, каждые несколько секунд выводящую сообщение. Реализация таймера будет осуществляться посредством функции `SetTimer` из библиотеки USER32.DLL, которой необходимо передать адрес stdcall вызываемой функции.

Листинг 3.4. Программа с stdcall-функциями (только секции кода и данных)

```
entry start
section '.data' data readable writeable

caption db 'First Win32 program',0
Message db 'Hello World!',0

counter dd 10
timerID dd ?
msg MSG

section '.code' code readable executable
```

```
ConcatStrings:
   push ebp
   mov ebp, esp
   push esi
   push edi
   ; param1 - [ebp+8]
   ; param2 - [ebp+12]
   ; param3 - [ebp+16]

   invoke lstrlen, [ebp+8]
   mov ecx, eax
   mov esi, [ebp+8]
   mov edi, [ebp+16]
   rep movsb

   invoke lstrlen ,[ebp+12]
   mov ecx, eax
   mov esi, [ebp+12]
   rep movsb

   pop edi
   pop esi
   leave
   retn 12

StdcallProcSample:

   push ebp
   mov ebp, esp
   sub esp, 8
   push ebx

   ; previous ebp - [ebp]
   ; return address [ebp+4]
   ; param1 - [ebp+8]
   ; param2 - [ebp+12]
   ; local var 1 - [ebp-4]
   ; local var 2 - [ebp-8]

   invoke lstrlen, [ebp+8]
   mov [ebp-4], eax
   invoke lstrlen, [ebp+12]
   add [ebp-4], eax
   inc dword [ebp-4]

   invoke GetProcessHeap
   mov [ebp-8], eax
   invoke HeapAlloc, eax, HEAP_ZERO_MEMORY, [ebp-4]
   mov [ebp-4], eax
```

```
    stdcall ConcatStrings, [ebp+8], [ebp+12], [ebp-4]

    invoke MessageBox, 0 , [ebp-4], [ebp+12], 0
    invoke HeapFree, [ebp-8], 0, [ebp-4]

    pop ebx
    leave
    retn 8

start:
    stdcall StdcallProcSample, Message, caption

    invoke ExitProcess, 0
```

В вышеприведённой программе есть две stdcall-функции. Первая из них производит конкатенацию (сложение) строк, а вторая осуществляет сложение двух переданных ей строк при помощи первой функции и выводит сообщение.

Функция `StdcallProcSample` получает в качестве параметров два указателя на строки. С помощью функции `lstrlen` сначала вычисляются длины этих строк. Функция `lstrlen` принимает указатель на строку в качестве единственного параметра и возвращает её длину. После чего выделяется буфер, равный сумме длин этих двух строк, с помощью функции `HeapAlloc`. Для выделения малых кусков памяти существует специальный набор функций, работающих с кучей; все они начинаются со слова `Heap`.

Функция `HeapAlloc` принимает три параметра: хендл кучи, в которой необходимо выделить буфер, параметры выделения и размер буфера. Для выделения памяти необходим хендл кучи, который надо указать первым параметром. Получить хендл первичной кучи процесса можно с помощью функции `GetProcessHeap`, которая не принимает параметров и возвращает хендл кучи, которая была создана системой при создании процесса. Если выделенный буфер необходимо обнулить, то в качестве параметров выделения следует указать `HEAP_ZERO_MEMORY`. При успешном результате функция `HeapAlloc` возвратит адрес выделенного буфера.

Первые две или три инструкции функции, создающей стековый фрейм, называются *прологом функции*; последние две являются *эпилогом функции*. Написание прологов и эпилогов функции – довольно-таки рутинное занятие. Специально для этого вместе с компилятором FASM поставляются макросы proc и endp, максимально упрощающие создание пролога и эпилога функции. Макросы proc и endp становятся доступными после включения файла win32a.inc.

Макрос proc создаёт пролог функции; при использовании этого макроса надо указать имя функции, а также имена параметров и имена локальных переменных. После использования макроса proc можно будет использовать имена параметров и имена локальных переменных вместо операндов `[ebp+/-N]`. Макрос endp генерирует эпилог функции и закрывает область действия имён параметров функции и локальных переменных, чтобы их нельзя было использовать в других местах программы. В листинге 3.5 приведён пример использования макросов proc и endp (в комментариях справа указано, на что будут заменены имена параметров и локальные переменные).

Листинг 3.5. Использование макросов proc и endp

```
proc MyFunc param1, param2
local lvar1:DWORD
local lvar2:DWORD

   mov eax, [param1]  ; mov eax, [ebp+8]
   mov ebx, [param2]  ; mov ebx, [ebp+12]
   mov [lvar1], eax   ; mov [ebp-4], eax
mov [lvar2], ebx   ; mov [ebp-8], ebx
   ret
endp
```

Наличие последней инструкции RET обязательно, т. к. без неё макрос endp работать не будет (ошибок при компиляции не возникнет, просто не будет сгенерирован эпилог функции и, следовательно, не будет возврата из функции). Макрос endp ищет последнюю команду RET и заменяет её на пару инструкций LEAVE/RETN.

Как бы выглядела программа, если бы мы использовали макрос proc? В листинге 3.6 приведена та же программа, что и в листинге 3.4, но с использованием макросов proc и endp (только секция кода).

Листинг 3.6. Программа с stdcall-функциями с использованием макросов proc и endp

```
section '.code' code readable executable

proc ConcatStrings str1, str2, str3
   push esi
   push edi
   invoke lstrlen, [str1]
   mov ecx, eax
   mov esi, [str1]
   mov edi, [str3]
   rep movsb

   invoke lstrlen ,[str2]
   mov ecx, eax
   mov esi, [str2]
   rep movsb

   pop edi
   pop esi
   ret
endp

proc StdcallProcSample str1, str2

local buff:DWORD
```

```
local hProcHeap:DWORD

   invoke lstrlen, [str1]
   mov [buff], eax
   invoke lstrlen, [str2]
   add [buff], eax
   inc dword [buff]

   invoke GetProcessHeap
   mov [hProcHeap], eax
   invoke HeapAlloc, eax, HEAP_ZERO_MEMORY, [buff]
   mov [buff], eax

   stdcall ConcatStrings, [str1], [str2], [buff]

   invoke MessageBox, 0 , [buff], [str2], 0
   invoke HeapFree, [hProcHeap], 0, [buff]

   ret
endp

start:
     stdcall StdcallProcSample, Message, caption

     invoke ExitProcess, 0
```

Как видно из листинга, при использовании макросов proc и endp код функций и программы в целом становится более читабельным и интуитивно понятным.

Вышеприведённые правила актуальны и для других соглашений вызовов: CDECL и PASCAL. (В соглашении CDECL параметры передаются в обратном порядке, и стек очищает вызывающая процедура, а в соглашении PASCAL параметры передаются в прямом порядке, и стек очищает вызываемая.)

Тем не менее программист не обязан действовать именно так. Он может использовать в качестве указателя фрейма стека другой регистр или вовсе ничего не использовать, но требования о неизменности регистров EBX, ESI, EDI и EBP остаются в силе, и если вы их задействуете, их надо будет сохранить и восстановить при выходе из функции.

3.1.8. Резюме

В разделе, посвященном программированию в Win32, мы познакомились с основами программирования в Win32 в третьем кольце. В практической части была написана простейшая программа Win32, а также было максимально подробно изучено соглашение вызова функций stdcall. В следующем разделе мы перейдем к более подробному изучению программирования в Win32 в третьем кольце. Полный исходный код программ находится на компакт-диске, прилагающемся к книге (папка part3).

3.2. Программирование в третьем кольце

В предыдущем разделе были изложены основы программирования на языке ассемблер в 32-битных системах Windows. Также было сказано, что основная часть исполняемых компонентов работает в третьем кольце защиты. Все программы, которые мы привыкли видеть на экране компьютера, работают на третьем уровне привилегий.

В разделе 3.2 будут изложены основные принципы создания различных типов программ, работающих на третьем уровне привилегий.

3.2.1. Общий обзор

При программировании в Windows как в третьем, так и в нулевом кольце очень часто приходится вызывать API-функции, предоставляемые операционной системой. Почти все API-функции третьего кольца вызываются по соглашению stdcall – согласно этой модели параметры функций передаются через стек в обратном порядке, при этом за очистку стека от параметров ответственна вызываемая функция. При частом вызове API-функций очень легко перепутать порядок передачи параметров. Для облегчения вызова API-функций вместе с компилятором FASM поставляется набор заголовочных файлов, содержащих макросы, при использовании которых вызов API-функций чрезвычайно облегчается.

FASM предоставляет два макроса для вызова API функций: stdcall и invoke. Оба макроса принимают указатель на функцию и параметры, передаваемые ей. Различие между ними только одно – способ передачи адреса вызываемой функции. Макросу stdcall необходимо передавать адрес функции, а макросу invoke – адрес, где хранится указатель на функцию. Например, чтобы вызвать функцию MessageBox в примере из предыдущего раздела с помощью макроса stdcall, необходимо написать следующее:

```
stdcall [MessageBox],0, Message, caption,0
```

А для вызова этой же функции с помощью макроса invoke необходимо написать следующее:

```
invoke MessageBox,0, Message, caption,0
```

Эта особенность иногда оказывается довольно-таки удобной, например когда адрес (или указатель на адрес) функции содержится в регистре.

Также вместе с компилятором FASM поставляется набор заголовочных файлов со значениями большинства констант, используемых при работе с API-функциями. Чтобы подключить полный набор заголовочных файлов с определениями всех констант, макросов и структур, достаточно подключить в программе файл win32a.inc для файлов с определениями для ANSI-версий функций – или win32w.inc для подключения файлов с определениями для UNICODE-версий функций.

Как уже было сказано в разделе 3.1, системы Windows поддерживают выполнение программ многих систем. «Родной» для Windows является подсистема Win32. Программы, работающие на третьем уровне привилегий в подсистеме Win32, бывают следующих типов:

1. Консольные программы.
2. GUI программы.

Главное отличие этих двух видов состоит в способе ведения диалога с пользователем. *Консольные программы* в этих целях пользуются консолью. *GUI-программы* ведут диалог с пользователем посредством графического интерфейса. Тем не менее большой разницы между этими двумя видами программ нет – отличия только формальные. Консольная программа может создать окно и посредством окна взаимодействовать с пользователем, а GUI-программа может создать консоль и взаимодействовать с пользователем через нее.

Но перед тем как перейти к описанию программирования вышеуказанных типов программ, необходимо изучить некоторые аспекты программирования в третьем кольце.

3.2.2. Работа с объектами

В Windows для работы с каким-либо объектом нужно получить его описатель, или хендл. *Описатель объекта* (далее – *хендл*) – это обычное число, которое задаёт номер элемента в некоторой таблице, описывающей объекты. В Windows есть два типа объектов: объекты ядра и пользовательские объекты. *Объектами ядра*, или *системными объектами* являются файлы, процессы, потоки, объекты синхронизации и т. д., т. е. все объекты, от которых прямо или косвенно зависит нормальная работа системы. *Объектами пользователя* в большинстве своём являются объекты графической подсистемы, а именно окна и элементы управления. Ядро системы не оперирует пользовательскими объектами и, по сути, не имеет о них никакого понятия.

У каждого процесса в Windows есть таблица хендлов, где описаны системные объекты, к которым процесс получил какой-либо доступ (как минимум на чтение). В этой таблице в каждом элементе содержатся имя объекта и уровень доступа к нему. В зависимости от уровня доступа к объекту возникает понятие уровня доступа хендла. К примеру, если хендл указывает на элемент таблицы, в котором описан файл с полным уровнем доступа, то его называют хендлом с полным доступом к файлу. Поскольку у каждого процесса своя таблица хендлов, то хендл, используемый в одном процессе (не забывайте, что хендл – это обычное число), в другом процессе может указывать на другой объект либо вовсе быть недействительным.

С пользовательскими объектами всё намного проще. Пользовательские объекты доступны всем процессам системы – хендлы являются общими для всех процессов системы.

3.2.3. Работа с файлами

Работа с файлами – один из ключевых аспектов при программировании в Windows. Большинство устройств в Windows имеют файловую структуру, а именно: файлы, каталоги, COM- и LPT-порты, многие периферийные устройства, все дисковые и недисковые устройства хранения информации, некоторые системные объекты. Даже если устройство не имеет файловую структуру, операционная система абстрагирует запросы к нему так, как будто запросы идут к файлам.

Всю основную работу с файлами осуществляют следующие API-функции, находящиеся в библиотеке KERNERL32.DLL:

1. `CreateFile`.
2. `ReadFile`.
3. `WriteFile`.

Функция `CreateFile` позволяет не только создавать файлы, но и открывать файловые объекты. Функция может открыть следующие объекты: файлы, пайпы, почтовые слоты, дисковые устройства, консоли, каталоги, а также любые устройства, имеющие имя и позволяющие работать с ними как с файлами.

Функция `CreateFile` принимает следующие параметры:

1. Имя файла (LPCTSTR).
2. Запрашиваемый доступ (DWORD).
3. Параметры общего доступа (DWORD).
4. Параметры безопасности (LPSECURITY_ATTRIBUTES).
5. Производимое действие (DWORD).
6. Дополнительные флаги и атрибуты (DWORD).
7. Файл-шаблон (HANDLE).

В скобках указаны названия типов параметров – именно те названия, которые были придуманы разработчиками операционной системы Windows. По сути, параметры, принимаемые функцией, являются обычными числовыми константами или указателями.

Первый параметр задаёт имя объекта (далее – файл), с которым будет идти работа. Строка может быть как ANSI, так и UNICODE. Второй параметр задаёт тип доступа к объекту; он может быть равен следующим значениям: GENERIC_READ (чтение) и GENERIC_WRITE (запись) либо их комбинацией. Третий параметр задаёт параметры общего доступа к файлу, а именно – какой доступ к открываемому файлу смогут получить другие программы. Если указана константа FILE_SHARE_READ, то другие процессы смогут только читать из файла; если указать FILE_SHARE_WRITE, то другие процессы смогут производить запись в файл. Четвёртый параметр задаёт атрибуты безопасности. В данном разделе они рассматриваться не будут; можно указать ноль, и будут применены атрибуты безопасности, задаваемые системой по умолчанию.

Пятый параметр задаёт операцию, производимую с файловым объектом. Можно передать следующие константы:

- CREATE_NEW – будет создан новый файл; если файл уже существует, функция вернёт ошибку.
- CREATE_ALWAYS – будет создан новый файл; если файл уже существует, то новый файл запишется поверх него (т. е. содержимое предыдущего файла будет потеряно).
- OPEN_EXISTING – файл будет открыт; если файла не существует, то функция вернёт ошибку.
- OPEN_ALWAYS – файл будет открыт в любом случае; если он не существует, то будет создан новый.

При открытии устройств или любых объектов, которые нельзя удалить или создать, необходимо указать OPEN_EXISTING в качестве пятого параметра – любая другая константа вызовет ошибку функции.

Шестой параметр задаёт дополнительные атрибуты открытия. Седьмой параметр позволяет указать хендл другого файла, на основе которого будет создан новый (используется совместно с действиями CREATE_NEW, CREATE_ALWAYS и OPEN_ALWAYS). Шестой и седьмой параметры рассматриваться в этом разделе не будут.

В случае удачи функция `CreateFile` вернёт описатель (хендл) объекта, имя которого было указано в первом параметре, в случае неудачи вернёт значение INVALID_HANDLE_VALUE (а именно -1 или 0FFFFFFFFh).

Поскольку операционные системы семейства WinNT (а речь идёт именно о них) являются сетевыми операционными системами, программа имеет возможность открывать файловые объекты на других системах, доступных через сеть. Для доступа к объектам на других системах необходимо указать имя объекта следующего формата: `\\`***`имя удалённого компьютера`***`\`***`имя объекта на удалённой системе`***. Указанный формат является форматом полного имени файла. Для открытия объектов в текущей системе можно указывать как полное имя объекта, так и неполное. Неполное имя объекта представляет собой привычное всем имя, например: `C:\folder\file.ext`. При использовании полного имени объекта можно указать имя текущего компьютера либо просто точку вместо имени, например: `\\.\D:\File.dat`. Можно получать доступ и к дисковым устройствам; например, если указать `\\.\PHYSICALDRIVE0`, то мы получим хендл, с помощью которого сможем работать с первым жёстким диском как с одним большим файлом. Кроме того, можно открывать логические диски и работать с ними как с одним большим файлом, например `\\.\D:`.

Помимо функции `CreateFile` есть функция `OpenFile`, которая позволяет только открывать файлы или устройства. Функция `OpenFile` является оболочкой вокруг функции `CreateFile`, к которой в результате всё и сводится так или иначе.

После открытия файла можно производить запись и чтение из него. Чтение из файла осуществляет функция `ReadFile`. Функция `ReadFile` принимает следующие параметры:

1. Хендл файла (HANDLE).
2. Указатель на буфер (LPVOID).
3. Количество читаемых байтов (DWORD).
4. Количество реально считанных байтов (LPDWORD).
5. Указатель на структуру, используемую при асинхронных операциях (LPOVERLAPPED).

Первый параметр задаёт хендл файла, с которым будет производиться операция чтения. Второй параметр задаёт указатель на буфер, в который будут сохранены считанные данные. Третий параметр задаёт указатель на переменную типа DWORD (двойное слово), в которую будет сохранено количество реально считанных байтов. Пятый параметр задаёт указатель на специальную структуру,

используемую при асинхронных операциях (они в этом разделе рассматриваться не будут).

Запись в файл производит функция `WriteFile`. Она принимает следующие параметры:

1. Хендл файла (HANDLE).
2. Указатель на буфер (LPVOID).
3. Количество записываемых байтов (DWORD).
4. Количество реально записанных байтов (LPDWORD).
5. Указатель на структуру, используемую при асинхронных операциях (LPOVERLAPPED).

Назначение параметров идентично тому, которое приводилось для функции `ReadFile`.

После завершения работы с файлом необходимо закрыть его хендл, чтобы другие программы могли работать с этим файлом. Функция `CloseHandle` производит закрытие хендла файла; она принимает один параметр – хендл файла.

Также помимо функций `WriteFile` и `ReadFile` есть более универсальная и гибкая функция – `DeviceIoControl`, которая позволяет работать не только с файлами, но и с устройствами.

Функция `CloseHandle` позволяет закрывать не только хендл файла, но и хендл любого системного объекта.

3.2.4. Обработка ошибок API-функций

При работе с API-функциями зачастую функции завершаются неудачей, и возникает необходимость узнать причину ошибки. Узнать код последней ошибки можно с помощью функции `GetLastError`. Функция не принимает никаких параметров и возвращает в регистре EAX код последней ошибки.

Операционная система Windows позволяет программе установить код последней ошибки. Функция `SetLastError` устанавливает код последней ошибки и принимает только один параметр – числовой код ошибки.

3.2.5. Консольные программы

Не следует путать консольные программы и старые DOS-программы – это совершенно разные типы программ. *Консольные программы* – это полноценные 32-битные программы, которые могут использовать все возможности операционной системы Windows. DOS-программы – это 16-битные программы режима реальных адресов, которые выполняются на виртуальной DOS-машине (процесс NTVDM.EXE); ничего общего с Windows они не имеют.

Главная особенность консольных программ в том, что при запуске к ним автоматически привязывается консоль, т. е. никаких действий со стороны программы не требуется – консоль создаётся автоматически. Если же GUI-приложению понадобится взаимодействовать с пользователем через консоль, оно должно будет создать консоль с помощью функции `AllocConsole`. По сути, это единственное отличие консольных программ от оконных.

Чтобы работать с консолью, необходимо получить её хендл. Есть несколько способов получения хендла консоли:

1. Через функцию `GetStdHandle`.
2. Через функцию `CreateFile`.

Исторически сложилось так: чтобы работать с консолью, надо иметь два хендла – один на чтение и один на запись. Функция `GetStdHandle` принимает один параметр; если он равен STD_INPUT_HANDLE, то функция возвращает хендл для чтения, а если STD_OUTPUT_HANDLE – хендл для записи. При использовании функции `CreateFile` необходимо задать только два параметра: имя и запрашиваемый доступ. Имя должно быть равно 'CON', а запрашиваемый доступ – GENERIC_READ (если надо получить хендл для чтения) или GENERIC_WRITE (если надо получить хендл для записи). Пятый параметр, который задаёт код действия, должен быть равен OPEN_EXISTING: любой другой код приведёт к ошибке.

Производить запись или чтение из консоли тоже можно двумя способами: через функции `ReadFile/WriteFile` или `ReadConsole/WriteConsole`. Между ними есть только одно отличие у функций `ReadConsole/WriteConsole` есть «функции-близнецы» `ReadConsoleW/WriteConsoleW`, которые позволяют работать с UNICODE-строками. Набор параметров функций `ReadConsole/WriteConsole` полностью идентичен параметрам функций `ReadFile/WriteFile`: хендл, указатель на буфер, количество записываемых байтов, указатель на переменную, в которую будет сохранено количество реально записанных/прочитанных байтов. Пятый параметр у функций `ReadConsole/WriteConsole` зарезервирован и должен быть равен нулю.

Функции записи в консоль `WriteConsole/WriteFile` производят немедленный вывод строки на консоль. Функции чтения из консоли `ReadConsole/ReadFile` ожидают нажатия пользователем клавиши **Enter** и заносят введённую строку в указанный программой буфер (символы новой строки и перевода каретки тоже заносятся в буфер).

3.2.6. GUI-программы

GUI-программы являются основой пользовательского интерфейса операционных систем Windows. Практически всё, что можно увидеть на экране операционной системы Windows, является результатом работы GUI-программ.

Основным понятием для GUI-программ является понятие *окна*. Практически все, что можно увидеть на экране, является окном: окно приложения, кнопка, поле ввода, многострочное поле ввода и т. д. Как уже было сказано, окна не являются объектами ядра системы, т. е., по сути, ядро системы об окнах ничего не знает, и вся реализация подсистемы GUI по большей части находится в пользовательском режиме в библиотеке user32.dll.

Каждое окно имеет два основных свойства: имя класса и оконную функцию. Класс задаёт основные свойства окна; оконная функция обрабатывает сообщения, отправляемые этому окну. Окна взаимодействуют с «окружающим миром» с помощью сообщений. Сообщения, присылаемые окну, выстраиваются в очереди. У каждого потока есть своя очередь сообщений.

Итак, начнём сначала. Для создания окна надо указать имя класса и ряд параметров – таких как положение, размеры, цвет фона окна и т. д. Самое главное при этом – имя класса. Поэтому перед созданием окна необходимо создать класс. Класс создаётся функцией `RegisterClass`. Функция принимает один параметр: указатель на структуру WNDCLASS. Далее описана структура WNDCLASS:

```
struct WNDCLASS
  style           dd ?
  lpfnWndProc     dd ?
  cbClsExtra      dd ?
  cbWndExtra      dd ?
  hInstance       dd ?
  hIcon           dd ?
  hCursor         dd ?
  hbrBackground   dd ?
  lpszMenuName    dd ?
  lpszClassName   dd ?
ends
```

Поле `style` задаёт базовые параметры окна. Описание всех возможных стилей не входит в задачу данного раздела; особую важность для нас имеют три следующих стиля:

- CS_HREDRAW – при указании этого стиля окно будет перерисовываться всякий раз, когда будет изменена ширина окна;
- CS_VREDRAW – при указании этого стиля окно будет перерисовываться всякий раз, когда будет изменена высота окна;
- CS_GLOBALCLASS – при указании этого стиля созданный класс будет глобален, т. е. другие приложения тоже смогут им воспользоваться.

Поле `lpfnWndProc` содержит указатель на оконную функцию. Поля `cbClsExtra` и `cbWndExtra` задают размеры дополнительных структур данных при создании класса; в наших программах эти поля будут равны нулю.

Поле `hInstance` задаёт хендл модуля, в котором находится оконная функция. При создании глобальных классов это поле должно быть задано. Поле `hIcon` задаёт хендл значка, который будет у окон созданного класса. Поле `hCursor` задаёт хендл курсора мыши, который будет использоваться при наведении мыши на окно.

Поле `hbrBackground` задаёт цвет фона окна. Поле `lpszMenuName` задаёт имя меню, которое будет использоваться в окнах создаваемого класса. Поле `lpszClassName` задаёт имя создаваемого класса.

Ключевым полем в данной структуре является указатель на оконную функцию. Оконная функция производит обработку сообщений, посылаемых данному окну. Оконная функция является функцией обратного вызова (CALLBACK); она вызывается самой системой. Функция должна соответствовать соглашению вызова stdcall. Оконная функция должна принимать четыре параметра:

1. Хендл окна.
2. Код сообщения.

3. Поле `wparam` сообщения.
4. Поле `lparam` сообщения.

Немаловажной при обработке сообщения является функция `DefWindowProc`. Она является универсальной оконной функцией. В оконной функции можно обрабатывать только специфичные для вашей программы сообщения – в остальных случаях можно передавать управление функции `DefWindowProc`, именно она и придаёт окну «классическое оконное поведение». Набор параметров у неё тот же, что и у оконной функции.

После создания класса окна можно приступить к созданию окна. Создание окна производит функция `CreateWindowEx`. Функция `CreateWindowEx` принимает следующие параметры:

1. Расширенный стиль окна.
2. Имя класса.
3. Имя окна (заголовок).
4. Стиль окна.
5. X-координата левого верхнего угла окна.
6. Y-координата левого верхнего угла окна.
7. Ширина окна.
8. Высота окна.
9. Хендл окна родителя.
10. Хендл меню.
11. Хендл модуля, ассоциированного с окном.
12. Указатель на структуру, передаваемую оконной функции при посылке сообщения WM_CREATE.

После создания окна можно приступить к обработке сообщений. Она происходит в бесконечном цикле получения сообщений.

Что такое сообщение? Фактически это структура с тремя полями:

1. Код сообщения.
2. Поле `wparam` сообщения.
3. Поле `lparam` сообщения.

Код сообщения задаёт действие, с которым оно ассоциировано: изменение размеров окна, перемещение, перерисовка, нажатие клавиши и т. д. Два поля wparam и lparam содержат данные сообщения. Например, при изменении размеров окна в этих полях указываются новые размеры окна.

Для получения сообщения из очереди сообщений потока можно использовать две функции – `GetMessage` и `PeekMessage`. Функция `GetMessage` принимает следующие параметры:

1. Указатель на структуру MSG для сохранения полученного сообщения.
2. Хендл окна.
3. Значение фильтра, задающее минимальный код сообщения.
4. Значение фильтра, задающее максимальный код сообщения.

Структура MSG содержит следующие поля:

```
struct MSG
  hwnd     dd ?
  message  dd ?
  wParam   dd ?
  lParam   dd ?
  time     dd ?
  pt       POINT
ends
```

Поле `hwnd` содержит хендл окна, которому было послано сообщение. Поле `message` содержит код сообщения. Поля `wparam` и `lparam` содержат данные, ассоциированные с этим сообщением. Поле `time` содержит время, когда сообщение было добавлено в очередь. Поле `pt` содержит координаты курсора в тот момент, когда сообщение было добавлено в очередь. Наиболее важными считаются первые четыре параметра; последние два иногда игнорируются при отправке сообщений.

Если при вызове функции `GetMessage` в качестве второго параметра ей передать хендл какого-либо окна, то она будет выбирать из очереди именно те сообщения, которые были адресованы данному окну. Если указать, ноль, то она выберет первое сообщение из очереди. С помощью третьего и четвёртого параметра можно задать диапазон кодов сообщений. Функция будет выбирать из очереди сообщения только из указанного диапазона. После получения сообщения функция `GetMessage` удаляет его из очереди.

Функция `PeekMessage` принимает пять параметров, причём первые четыре идентичны параметрам функции `GetMessage`. Пятый параметр задаёт действие, производимое после получения сообщения из очереди: если он равен PM_NOREMOVE, то сообщение не будет удалено из очереди, а если равен PM_REMOVE, то будет удалено.

Между `PeekMessage` и `GetMessage` есть одно главное отличие: если очередь сообщений пуста, то функция `GetMessage` ждёт появления сообщения, в то время как `PeekMessage` сразу же завершается. Если функция `PeekMessage` вернула ноль, значит, очередь сообщений была пуста (либо не было ни одного подходящего сообщения в очереди). Если же функция `GetMessage` возвращает ноль, это может значить только одно – она получила сообщение WM_QUIT – в этом случае необходимо завершить цикл обработки сообщений.

3.2.7. Динамически подключаемые библиотеки

Ещё одним немаловажным компонентом третьего уровня привилегий являются *динамически подключаемые библиотеки* (DLL, Dynamic Link Library). Динамически подключаемые библиотеки, по сути, являются такими же программами, как и обычные EXE-файлы. Есть только одно отличие: наличие *таблицы экспорта*. С помощью таблицы экспорта задаются имена и адреса экспортируемых функций.

DLL, так же как и обычный исполняемый файл, имеет точку входа. Код, выполняющийся в точке входа DLL, обычно подготавливает данные для нормальной работы экспортируемых функций. Управление на точку входа передаётся всякий раз, когда она загружается в память какого-либо процесса.

Благодаря использованию DLL экономится физическая память. Экономия достигается за счёт того, что область памяти, в которую загружается DLL, помечается как общая для нескольких процессов. После того как какой-либо другой процесс захочет загрузить эту же DLL в своё адресное пространство, физически DLL не загружается в память – вместо этого в виртуальную память процесса проецируется общая область памяти, в которую загружена эта DLL. Например, DLL занимает в памяти 64 Кб; при её загрузке в память трёх процессов расходуется 64 Кб физической памяти и 192 Кб виртуальной (по 64 Кб в каждом процессе).

Поскольку память общая для нескольких процессов, любое изменение данных DLL в одном процессе может привести к непредсказуемым последствиям в других. Поэтому область памяти, в которую загружаются DLL, помечается как копируемая при записи – иными словами, при первой операции записи в её память страница копируется и становится индивидуальной для процесса, осуществившего запись, т. е. перестаёт быть общей. Таким образом, при изменении каких-либо данных библиотеки в виртуальной памяти одного процесса изменения не отражаются на памяти других процессов. На рис. 3.2 приведён пример расположения DLL в физической и виртуальной памяти.

В этом примере библиотека загружена в три процесса, но, как видно, в физической памяти находится только одна копия DLL. В процессе № 3 произошла операция записи на третью страницу; после этого в физической памяти была создана копия этой страницы, и третья страница для процесса № 3 стала индивидуальной. Изменение данных на третьей странице не отразилось на памяти других процессов.

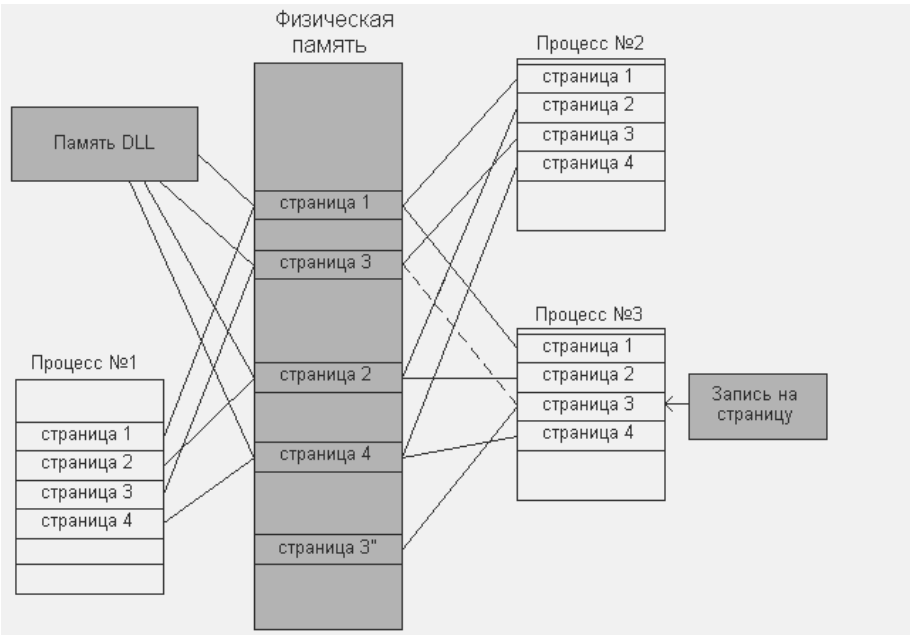


Рис. 3.2. Пример расположения DLL в физической и виртуальной памяти

Для динамических библиотек очень важной становится секция релокейшенов. Если дело касается обычных исполняемых файлов, которые загружаются в память одними из первых, секцией релокейшенов можно пренебречь, т. к. в 99,99% случаев адрес, по которому будет загружаться исполняемый файл, будет свободен. При загрузке DLL высока вероятность, что адреса памяти, которые прописаны в заголовке DLL, будут заняты. При загрузке DLL загрузчик в первую очередь проверяет занятость адреса, который прописан в заголовке DLL; если он занят, то DLL загружается в другую свободную область памяти. После чего загрузчик, пользуясь секцией релокейшенов, подправляет все команды в программе, которые обращаются к памяти, для того чтобы в работе программы не было ошибок: ведь если DLL будет загружена по другому адресу, значит, и данные, используемые функциями из DLL, тоже загрузятся по другому адресу.

Есть два способа импорта функций из DLL: статический и динамический. Статический метод импортирования функций был использован нами в примере из раздела 3.1 при использовании функций `MessageBox` из библиотеки user32.dll и `ExitProcess` из kernel32.dll. При статическом импорте функции в таблице импорта надо прописать имя DLL, в которой находится функция, и имя функции (реже – индекс функции). Для создания таблицы импорта пакет FASM предоставляет программисту макросы library и import – они скрывают от нас все тонкости создания таблицы импорта.

При динамическом импорте функций загрузка DLL и получение адреса нужной функции осуществляются посредством двух функций: `LoadLibrary` и `GetProcAddress`. У каждого способа импорта есть свои плюсы и минусы. При статическом импорте загрузчик делает всю работу за нас. Динамический импорт функций требует от программиста больше работы, но DLL можно загружать только по мере надобности и, как только она станет ненужной, выгружать её из памяти при помощи функции `FreeLibrary`.

Функция может экспортироваться двумя способами: по имени и по ординалу (индексу). При экспорте по имени функции в таблице экспорта прописывается имя функции, при экспорте по ординалу – номер функции.

Функция `LoadLibrary` производит загрузку DLL в память процесса. Функция принимает один параметр – имя файла DLL, возвращает хендл модуля DLL. Хендл модуля отличается от всех других хендлов в системе: фактически это базовый адрес, по которому загружен модуль, в нашем случае – DLL.

Функция `GetProcAddress` получает адрес экспортируемой функции. Функция принимает два параметра: хендл модуля и имя функции либо её номер. Если вместо указателя на имя функции передан номер функции, то он не должен превышать значения 65536. В случае успеха функция `GetProcAddress` возвращает указатель на экспортируемую функцию.

Функция `FreeLibrary` принимает в качестве единственного параметра хендл DLL и осуществляет выгрузку DLL из памяти процесса.

В отличие от обычных программ точка входа DLL должна указывать на специальную функцию, которая также будет вызвана при выгрузке DLL. Функция должна отвечать соглашению stdcall и иметь три параметра:

1. Хендл DLL.
2. Произошедшее событие.
3. Дополнительная информация о событии.

События могут быть следующие:

1. DLL_PROCESS_ATTACH – библиотека была загружена в память процесса.
2. DLL_THREAD_ATTACH – в текущем процессе создан новый поток.
3. DLL_THREAD_DETACH – в текущем потоке уничтожен поток.
4. DLL_PROCESS_DETACH – библиотека будет выгружена из памяти процесса.

Интерпретация третьего параметра зависит от произошедшего события. Если произошло событие DLL_PROCESS_ATTACH и третий параметр равен нулю, то происходит динамическая загрузка DLL; если не равен нулю – то статическая. Если произошло событие DLL_PROCESS_DETACH и третий параметр равен нулю, то происходит выгрузка DLL при помощи функции `FreeLibrary`; если не равен нулю, то выгрузка DLL происходит из-за завершения процесса.

3.2.8. Обработка исключений в программе

Для обработки исключений в третьем кольце защиты Windows предоставляет механизм SEH (Structured Exception Handling). Каждая программа может определить свой собственный обработчик исключений для обработки ошибок, возникающих в ходе её выполнения.

При возникновении исключений, обработку которых можно предоставить коду третьего кольца, контроль передается текущему SEH-обработчику. Примером таких исключений могут быть следующие: ошибка деления (#DE), исключение общей защиты (#GP), страничное нарушение (#PF).

Механизм SEH спроектирован таким образом, что можно определять несколько обработчиков, в результате чего возникает понятие цепочки обработчиков SEH. Если ни один обработчик не справляется с исключением, то он передаёт управление следующему в цепочке. В конце концов, если последний обработчик в цепочке не справляется с обработкой, он передаёт управление функции `UnhandledExceptionFilter`, которая является стандартным обработчиком, в зависимости от типа ошибки просто выводит сообщение об ошибке и завершает приложение.

Использование механизма SEH для обработки ошибок в программе может быть полезным в следующих случаях:

1. При использовании непроверенного кода, написанного другими людьми.
2. При обращении (чтение/запись) к области памяти, которая может быть перемещена без предупреждения – например, во время исследования системных областей памяти (которыми управляет сама система) или областей памяти, которые могут быть закрыты другими процессами и/или потоками;
3. При использовании указателей на файлы, которые могут быть разрушены или иметь неопределенный формат.
4. При непредвиденных ошибках.

Цепочка обработчиков SEH оформлена в виде односвязного списка. Элементом в этом односвязном списке является структура _EXCEPTION_REGISTRATION:

```
struct _EXCEPTION_REGISTRATION
 prev      dd      ?
 handler   dd      ?
ends
```

Поле prev содержит адрес следующей структуры _EXCEPTION_REGISTRATION, а поле handler – адрес обработчика. Чтобы определить свой обработчик исключений, надо добавить структуру _EXCEPTION_REGISTRATION с указателем на функцию-обработчик в начало цепочки обработчиков SEH.

Теперь зададимся вопросом: как же найти указатель на начало цепочки обработчиков SEH? У каждого потока есть область данных TEB (Thread Environment Block) – блок информации потока, который содержит параметры и данные потока. В TEB первый параметр DWORD выступает в качестве указателя на структуру _EXCEPTION_REGISTRATION, которая является начальной в цепочке обработчиков SEH. Сегментный регистр FS является указателем на блок TEB; вернее, в него загружен селектор, который описывает область памяти, где находится TEB. Поэтому адрес структуры _EXCEPTION_REGISTRATION, которая описывает первый обработчик в цепочке SEH, находится в [FS:0]. Например, следующий код добавляет новый обработчик в цепочку SEH:

```
push        exception_handler
push        dword [FS:0]
mov         [FS:0],esp
```

Первые две команды push создают в стеке два DWORD-значения, которые являются указателями на структуру _EXCEPTION_REGISTRATION, описывающую текущий (теперь уже предыдущий) обработчик и указатель на новый обработчик; таким образом, в стеке мы создали структуру _EXCEPTION_REGISTRATION, описывающую новый обработчик. После в ячейку [FS:0] мы заносим указатель (на структуру), который содержится в регистре ESP.

Вопрос: зачем определять структуру _EXCEPTION_REGISTRATION в стеке – почему нельзя объявить её как глобальную переменную? Всё очень просто: код ядра, который управляет SEH, проверяет, где находится экземпляр структуры _EXCEPTION_REGISTRATION. Если эта структура находится не в стеке – он вызывает аварийное исключение.

Функция-обработчик, как ни странно, должна отвечать соглашению вызова cdecl. Отличие соглашения cdecl от stdcall заключается в том, что очищать стек от параметров должна не вызываемая функция, а вызывающий её код. Функция-обработчик принимает четыре параметра:

1. Указатель на структуру с информацией об исключении (EXCEPTION_RECORD).
2. Указатель на узел цепочки обработчиков, из которого вызван этот обработчик.
3. Указатель на структуру, содержащую контекст потока в момент возникновения исключения (CONTEXT).
4. Указатель на прерванный фрейм в случае вложенных исключений (EXCEPTION_RECORD).

Из параметров нам интересны только первый и третий. Третий указывает на структуру CONTEXT, которая содержит значения всех регистров в момент возникновения исключения. Структура EXCEPTION_RECORD представляет собой блок информации, описывающий исключение, которое возникло в ходе выполнения потока. Формат этой структуры таков:

```
struct EXCEPTION_RECORD
  ExceptionCode dd ?
  ExceptionFlags dd ?
  ExceptionRecord dd ?
  ExceptionAddress dd ?
  NumberParameters dd ?
  ExceptionInformation dd EXCEPTION_MAXIMUM_PARAMETERS dup (?)
ends
```

Поле ExceptionCode содержит код исключения. В табл. 3.1 приведены наиболее часто возникающие исключения.

Таблица 3.1. Наиболее частые исключения

Код исключения	Описание
EXCEPTION_ACCESS_VIOLATION	Попытка чтения/записи области памяти без соответствующих разрешений
EXCEPTION_BREAKPOINT	Достигнута точка останова
EXCEPTION_ILLEGAL_INSTRUCTION	Попытка выполнения неподдерживаемой инструкции
EXCEPTION_IN_PAGE_ERROR	Попытка обращения к отсутствующей странице, если её невозможно загрузить
EXCEPTION_INT_DIVIDE_BY_ZERO	Попытка целочисленного деления на ноль
EXCEPTION_NONCONTINUABLE_EXCEPTION	Попытка возобновления выполнения потока после возникновения «непродолжаемого» исключения
EXCEPTION_PRIV_INSTRUCTION	Попытка выполнения привилегированной инструкции
EXCEPTION_SINGLE_STEP	В процессе трассировки была выполнена одна инструкция
EXCEPTION_STACK_OVERFLOW	Переполнение стека

Поле ExceptionFlags предназначено для определения того, является ли данное исключение продолжаемым. В случае, если это не так, будет установлен флаг EXCEPTION_NONCONTINUABLE. Попытка возобновить выполнение потока после возникновения «непродолжаемого» исключения приведёт к возникновению исключения с кодом EXCEPTION_NONCONTINUABLE_EXCEPTION. Случаи возникновения вложенных исключений в данном разделе рассматриваться не будут.

ExceptionRecord служит для организации цепочки из структур EXCEPTION_RECORD. Это свойство используется применительно к вложенным исключениям, когда исключения происходят во время их обработки.

Поле ExceptionAddress содержит информацию о месте возникновения исключения.

В поле NumberParameters содержится количество параметров, ассоциированных с данным исключением и хранящихся в массиве ExceptionInformation. Это позволяет передавать обработчикам дополнительную информацию об исключении. Например, в случае с EXCEPTION_ACCESS_VIOLATION в первом элементе данного массива передаётся флаг операции (чтение/запись), а второй элемент содержит виртуальный адрес, к которому производилось обращение.

Задача обработчика исключений в основном заключается в оценке типа и причины исключения – если обработчик может справиться с ситуацией, то устраняет эту причину.

После того как исключение обработано, обработчик должен вернуть значение ExceptionContinueExecution (значение 0). В случае если решением проблемы является завершение потока или завершение выполнения функции, обработчик должен инициировать так называемую «раскрутку». Она необходима в случае наличия вложенных защищаемых блоков кода и необходимости освободить ресурсы при возникновении ошибки в одном из них. В данном разделе процесс «раскрутки» рассматриваться не будет.

Если обработчик не может обработать исключение, он должен вернуть значение ExceptionContinueSearch (значение 1). После этого система вызовет следующий обработчик.

Более подробно работа механизма SEH будет описана в параграфе 3.2.9.4 на конкретном примере.

3.2.9. Практика

Чистая теория – всего лишь бесполезный набор знаний. Необходимо закрепить все полученные знания на практике. Далее будут приведены только наиболее важные участки кода; полный исходный код программ, приведённых ниже, можно найти на компакт-диске в папке part3.

3.2.9.1. Создание консольного приложения

В качестве примера консольного приложения напишем приложение, которое вычисляет какую-нибудь формулу. Формула будет следующая: $(a + 3) * 6 + a * (100 - a)$, число a будет вводиться с консоли.

Вначале надо получить необходимые хендлы, вывести сообщение и получить введённое число (листинг 3.7).

Листинг 3.7. Получение хендлов и вывод сообщения в консоль

```
invoke GetStdHandle, STD_INPUT_HANDLE
mov [InputHandle], eax

invoke GetStdHandle, STD_OUTPUT_HANDLE
mov [OutputHandle], eax
```

```
invoke WriteConsole, [OutputHandle], inputa, 8,Writed,0
invoke ReadConsole, [InputHandle], a_str, MAX_VALUE_LENGTH, Readed,0
```

Функция `ReadConsole` будет ждать ввода какого-либо текста и нажатия клавиши **Enter**. После нажатия **Enter** введённая строка будет скопирована в буфер, на который указывает второй параметр, переданный функции `ReadConsole` – в данном случае в буфер, на который указывает метка `a_str`.

После введения числа надо преобразовать введённую строку в число (листинг 3.8).

Листинг 3.8. Ввод числа

```
mov edi, a_str
call GetZSLength
sub eax, 2
mov ebx, eax
add eax, a_str
mov word [eax], 0

mov esi, a_str
call STR_to_DWORD_EX
```

С помощью функции `GetZSLength` получаем длину строки. После получения длины строки производится обнуление двух последних символов, т. к. функция `ReadConsole` заносит в буфер символы перевода каретки и начала новой строки. После обнуления двух последних символов происходит преобразование строки в число. Функция `STR_to_DWORD_EX` получает указатель на строку через регистр ESI, а результат преобразования заносит в регистр EAX.

Следующий шаг – это вычисление формулы (листинг 3.9).

Листинг 3.9. Вычисление формулы

```
push eax
add eax, 3
imul eax, 6
mov ebx, eax  ; ebx = (a+3)*6
mov edx, 100  ;
pop eax
sub edx, eax
imul eax, edx ; eax = a*(100-a)
add eax, ebx  ; eax = res
```

Прежде всего надо временно сохранить число в стеке, чтобы позже им воспользоваться. Сначала вычисляется первая часть формулы, потом вторая.

После вычисления формулы надо вывести результат на консоль; для этого необходимо преобразовать результат в строку (листинг 3.10).

Листинг 3.10. Преобразование результата в строку и вывод в консоль

```
mov esi, a_str
mov ebx, 10
call dword_to_STR
```

```
mov edi, message
call GetZSLength

invoke WriteConsole, [OutputHandle], message, eax,Writed,0
invoke ReadConsole, [InputHandle], a_str, 1,0,0
```

Преобразование числа в строку происходит с помощью функции `dword_to_STR`, она принимает число в регистре EAX и указатель на буфер в регистре ESI; строка-результат сохраняется в буфере. После вывода результата происходит вызов функции `ReadConsole`. Он нужен для ожидания нажатия клавиши Enter, чтобы программа не закрылась сразу же после вывода результата.

3.2.9.2. Создание оконного приложения

В качестве примера оконного приложения напишем программу, создающую окно и две кнопки на нём: нажатие первой кнопки выводит на экран сообщение, нажатие второй завершает программу.

Первый шаг – это регистрация класса нашего окна. Соответствующий код приведён в листинге 3.11.

Листинг 3.11. Регистрация нового класса окна

```
invoke  GetModuleHandle,0

mov  [hInst], eax
mov  [wc.style], CS_HREDRAW + CS_VREDRAW + CS_GLOBALCLASS
mov  [wc.lpfnWndProc],  WndProc
mov  [wc.cbClsExtra], 0
mov  [wc.cbWndExtra], 0
mov  [wc.hInstance], eax

invoke  LoadIcon,0,IDI_APPLICATION
mov  [wc.hIcon], eax

invoke  LoadCursor,0, IDC_ARROW
mov  [wc.hCursor], eax

mov  [wc.hbrBackground], COLOR_BACKGROUND+1
mov  dword [wc.lpszMenuName], 0
mov  dword [wc.lpszClassName], szClassName

invoke  RegisterClass, wc
```

В самом начале происходит получение хендла текущего модуля; он пригодится при заполнении полей структуры, описывающей класс. Хендл модуля – это фактически адрес, по которому он загружен в память. После заполнения всех полей структуры `WNDCLASS` происходит вызов функции `RegisterClass` и передача ей указателя на структуру `WNDCLASS`. Получение хендлов курсора и иконки осуществляется при помощи соответствующих функций; их использование тривиально.

После регистрации класса надо создать окно (листинг 3.12).

Листинг 3.12. Создание окна

```
invoke CreateWindowEx,0,szClassName,szTitleName,\
     WS_OVERLAPPEDWINDOW, 50,50, 300, 250,\
     0,0,[hInst],0
mov  [main_hwnd], eax

invoke CreateWindowEx, 0, button_class, AboutTitle,\
     WS_CHILD, 50, 50, 200, 50, [main_hwnd],0,[hInst],0
mov  [AboutBtnHandle], eax

invoke CreateWindowEx, 0, button_class, ExitTitle,\
     WS_CHILD, 50, 150, 200, 50, [main_hwnd],0,[hInst],0
mov  [ExitBtnHandle], eax
```

При создании главного окна был указан стиль `WS_OVERLAPPEDWINDOW`, который применяется для создания стандартных окон; т. е. у окна будут заголовок, три стандартные кнопки в правом верхнем углу, это окно можно будет перемещать и изменять его размеры. При создании кнопок в качестве имени класса был передан указатель на строку `BUTTON` (см. полный исходник). Класс `BUTTON` имеют все обычные кнопки. Также при создании кнопок был указан стиль `WS_CHILD` и указан хендл главного окна в качестве родительского.

Следующий шаг (листинг 3.13) – это обновление окон и начало цикла обработки сообщений.

Листинг 3.13. Цикл получения сообщений

```
invoke    ShowWindow,[main_hwnd],SW_SHOWNORMAL
invoke    UpdateWindow,[main_hwnd]
invoke    ShowWindow, [AboutBtnHandle],SW_SHOWNORMAL
invoke    ShowWindow, [ExitBtnHandle],SW_SHOWNORMAL

msg_loop:
     invoke   GetMessage, msg,0,0,0

     cmp    eax, 0
     je     end_loop

     invoke TranslateMessage, msg
     invoke DispatchMessage, msg

     jmp    msg_loop

end_loop:
```

Функция `ShowWindow` позволяет скрыть или показать окно; её использование тривиально. Описание данной функции здесь приводиться не будет. Функция `UpdateWindow` производит обновление окна. После обновления окна и вывода

кнопок начинается цикл обработки сообщений. В цикле используются функции GetMessage, TranslateMessage и DispatchMessage. Назначение функции GetMessage было рассмотрено нами выше. Функция TranslateMessage переводит сообщения о нажатиях клавиш в сообщения о вводе строковых данных. Назначение функции DispatchMessage на первый взгляд кажется странным – она вызывает функцию-обработчик сообщений окна, которому пришло сообщение. Почему мы не можем вызвать её сами и прибегаем к помощи «посредника»? Дело в том, что окон может быть много, у некоторых окон (кнопок, полей ввода и т. д.) чаще всего обработчиком является функция, назначенная системой. Функция DispatchMessage делает всю «грязную» работу за нас.

Осталось только написать оконную функцию. Оконная функция должна соответствовать соглашению stdcall. Макрос proc позволяет объявить stdcall-функцию. Макросу надо передать имя функции и имена параметров. Чтобы обозначить конец функции, нужно использовать макрос endp. Макрос proc позволяет в теле функции использовать символьные имена параметров вместо их адресов в стеке. Это избавляет нас от вычисления расположения параметров в стеке. Макрос proc скрывает от нас все тонкости создания stdcall-функций.

Далее в листинге 3.14 приведена оконная функция.

Листинг 3.14. Функция обработки сообщений окна

```
proc WndProc hwnd, wmsg, wparam, lparam

     pushad
     cmp  [wmsg], WM_DESTROY
     je .wmdestroy
     cmp  [wmsg], WM_COMMAND
     jne  .default
     mov  eax, [wparam]
     shr  eax, 16
     cmp  eax, BN_CLICKED
     jne  .default
     mov  eax, [lparam]
     cmp  eax, [AboutBtnHandle]
     je  .about
     cmp  eax, [ExitBtnHandle]
     je  .wmdestroy

.default:
     invoke  DefWindowProc, [hwnd],[wmsg],[wparam], [lparam]
     jmp  .finish

.about:
     invoke MessageBox, 0, AboutText, szTitleName,0
     jmp .finish

.wmdestroy:
```

```
        invoke  ExitProcess,0
.finish:
        mov [esp+28], eax
        popad
        ret
endp
```

Ключевой момент в вышеприведённой функции – это обработка сообщения WM_COMMAND. Для обработки щелчков по кнопкам необходимо переопределить обработчик класса BUTTON либо создать свой класс, похожий на BUTTON, но более простой способ определить щелчки по кнопкам – это обработка сообщений WM_COMMAND. Все дочерние окна посылают своему родительскому окну сообщение WM_COMMAND, когда в них происходят определённые события. При щелчке по кнопке класса BUTTON родительскому окну посылается сообщение WM_COMMAND с кодом BN_CLICKED в старшей части параметра wparam и хендлом кнопки в параметре lparam. В нашем случае при щелчке по кнопке About происходит вывод сообщения, а при щелчке по кнопке Exit – завершение программы.

3.2.9.3. Создание DLL

В качестве примера динамически подключаемой библиотеки напишем DLL, которая будет содержать функцию, вычисляющую некоторую формулу. Перепишем консольное приложение, созданное нами в параграфе 3.2.9.1, так, чтобы оно использовало функцию из DLL. Будем осуществлять динамический импорт функции.

Далее в листинге 3.15 приведён код DLL.

Листинг 3.15. Секция кода DLL

```
section '.code' code readable executable

proc DllEntryPoint hinstDLL,fdwReason,lpvReserved
   mov  eax, TRUE
   ret
endp

proc CalcValue Value
   push ebx
   push edx

   push eax
   add eax, 3
   imul eax, 6
   mov ebx, eax  ; ebx = (a+3)*6

   mov edx, 100  ;
   pop eax
   sub edx, eax
   imul eax, edx ; eax = a*(100-a)

   add eax, ebx  ; eax = res
```

```
    pop edx
    pop ebx
    ret
endp

section '.edata' export data readable

  export 'DLL_sample.DLL',\
    CalcValue,'CalcValue'

section '.reloc' fixups data discardable
```

Как видно, ничего сложного в написании DLL нет. Ключевой момент в коде DLL – это таблица экспорта. Макрос export максимально упрощает процесс создания таблицы экспорта.

Теперь осталось только изменить код программы, так чтобы она вызывала функцию из DLL (листинг 3.16).

Листинг 3.16. Код, использующий функцию из DLL

```
invoke WriteConsole, [OutputHandle], inputa, 8,Writed,0
invoke ReadConsole, [InputHandle], a_str, MAX_VALUE_LENGTH, Readed,0

mov edi, a_str
call GetZSLength
sub eax, 2
mov ebx, eax
add eax, a_str
mov word [eax], 0

mov esi, a_str
call STR_to_DWORD_EX
push eax

invoke LoadLibrary, DLL_name          ; eax = DLL handle
invoke GetProcAddress, eax, FUNC_name ; eax = func address
mov ebx, eax                          ; ebx = eax
pop eax                               ; eax = value
stdcall  ebx, eax                     ; call func

mov esi, a_str
mov ebx, 10
call dword_to_STR


mov edi, message
call GetZSLength

invoke WriteConsole, [OutputHandle], message, eax,Writed,0
invoke ReadConsole, [InputHandle], a_str, 1,0,0
```

Сначала происходит загрузка DLL с помощью функции `LoadLibrary`, потом получение адреса функции `CalcValue`. Потом с помощью макроса stdcall осуществляется вызов функции `CalcValue`, указатель на которую содержится в регистре EBX.

3.2.9.4. Использование SEH

В демонстрационных целях напишем небольшую программу, которая будет создавать свой собственный обработчик исключения и генерировать две ошибки: ошибку деления на ноль и ошибку доступа к памяти.

В листинге 3.17 приведён код функции обработчика исключений.

Листинг 3.17. Обработчик исключений

```
proc ExceptionHandler c ExceptionRecord, EstablisherFrame, ContextRecord,
DispatcherContext

   push ebx
   push esi

   mov  ebx, [ExceptionRecord]
   virtual at ebx
    .ebx EXCEPTION_RECORD
   end virtual

   mov  esi, [ContextRecord]
   virtual at esi
    .esi CONTEXT
   end virtual

   cmp  [.ebx.ExceptionCode], EXCEPTION_ACCESS_VIOLATION
   je  .AccessViolation
   cmp  [.ebx.ExceptionCode], EXCEPTION_INT_DIVIDE_BY_ZERO
   je  .DivideError

    .ContinueSearch:
   mov  eax, ExceptionContinueSearch
   jmp .exit

    .AccessViolation:
   invoke MessageBox, 0, szAccessViolationError, szTitle, 0
   add [.esi.regEip], 2
   mov eax, ExceptionContinueExecution
   jmp .exit

    .DivideError:
   invoke MessageBox, 0, szDivideByZeroError, szTitle, 0
   add [.esi.regEip], 2
   mov eax, ExceptionContinueExecution

    .exit:
```

```
    pop esi
    pop ebx
    ret
endp
```

Символ c после имени функции означает, что макрос proc будет создавать функцию, отвечающую соглашению cdecl. Как и любая функция обратного вызова, функция-обработчик исключения не должна изменять значения регистров EBX, ESI и EDI (желательно не должна менять значения ни одного регистра).

Вначале функция помещает указатели на информацию об исключении и контекст потока в регистры EBX и ESI соответственно, после чего анализирует тип исключения и в зависимости от него изменяет значение регистра EIP таким образом, что инструкция, вызывавшая исключение, пропускается. В обоих случаях регистр EIP увеличивается на 2 (зачем – будет пояснено позже).

Коды ошибок, использованные в вышеприведённом обработчике, а также приведённые в табл. 3.1, определены в файле DDK\ntdefs.inc.

Обработчик исключений написан – остаётся только зарегистрировать его и протестировать, сгенерировав исключения (листинг 3.18).

Листинг 3.18. Регистрация обработчика исключений

```
start:
    push ExceptionHandler
    push dword [fs:0000h]
    mov  dword [fs:0000h], esp

    xor eax, eax
    div eax

    xor eax, eax
    mov eax, [eax]

    pop eax
    mov [fs:0000h], eax
    pop eax

    invoke MessageBox, 0, szMessage, szTitle, 0
    invoke ExitProcess, 0
```

Сначала регистрируется обработчик, потом генерируются два исключения: деление на ноль и обращение к нулевому адресу (нулевой адрес в Win32 является недопустимым). Затем обработчик удаляется из цепочки обработчиков.

3.2.9.5. Отладка приложений

При разработке программ на ассемблере практически невозможно обойтись без отладчика. Отладка программ, работающих без операционной системы, была почти невозможной – разве что некоторые виртуальные машины (QEMU и Bochs) предоставляли для этого некоторые базовые возможности. При разработке программ под Windows ситуация коренным образом изменилась. Для отладки приложений

под операционными системами Win32 существует широкий выбор отладчиков приложений, работающих в третьем кольце; наиболее распространёнными из них являются OllyDebug, IDA Pro и WinDBG.

Отладчики OllyDebug и WinDBG являются полностью бесплатными, а за использование отладчика (по совместительству и дизассемблера) IDA Pro придётся заплатить. Впрочем, это единственный его недостаток, который полностью компенсируется широким функционалом. Отладчик WinDBG можно бесплатно скачать на официальном сайте компании Microsoft; также его можно найти на компакт-диске, прилагающемся к данной книге (папка debuggers). Там же вы найдете и отладчик OllyDebug.

При отладке программ иногда очень полезной бывает команда INT3. В отличие от команды вызова прерывания INT n, команда INT3 имеет короткий однобайтовый опкод. При использовании более длинной команды INT 3 обработчик может быть и не вызван из-за ограничений безопасности. При использовании короткой команды INT3 обработчик отладочного исключения будет вызван в любом случае – разумеется, если он определён, т. е. если в текущий момент времени программа находится под отладкой. Команда INT3 может упростить некоторые действия; например, можно её поставить в критических местах, и отладчик остановит выполнение программы именно в том месте, где встречается команда INT3.

При отладке обычной программы отладчик автоматически останавливает её выполнение в точке входа, после чего у нас есть возможность поставить точки останова во всех необходимых нам местах, поэтому при отладке обычных приложений в принципе можно и обойтись без использования команды INT3. При отладке динамически подключаемых библиотек перехватить момент загрузки нашей DLL не так просто, поэтому распространённым приёмом является вставка инструкции INT3 в точку входа отлаживаемой библиотеки. В результате достаточно взять под отладку приложение, использующее нашу DLL, и отладчик сразу остановит её выполнение после того, как управление будет передано точке входа нашей библиотеки.

3.2.10. Резюме

В данном разделе мы ознакомились с ключевыми понятиями и концепциями, необходимыми при программировании в третьем кольце в Win32. Тем не менее данный материал является лишь малой частью от общего объёма. Не рассмотренными остались такие аспекты программирования в третьем кольце, как: взаимодействие с системными объектами, графической подсистемой, создание многопоточных приложений, взаимодействие с COM- и OLE-объектами, работа с реестром, виртуальной памятью, кучей и т.д. В следующем разделе речь пойдёт о программировании в нулевом кольце в системах Win32.

3.3. Программирование в нулевом кольце

В разделе 3.2 были изложены основы программирования в третьем кольце на 32-битных системах Windows. В данном разделе мы рассмотрим основные принципы создания программ, работающих на нулевом уровне привилегий.

На нулевом уровне привилегий работает код ядра Windows; работа на нулевом уровне привилегий также называется работой в режиме ядра Windows. Главными исполняемыми компонентами ядра Windows являются драйверы – именно они составляют большую часть кода нулевого кольца.

3.3.1. Службы

Прежде чем начнётся разговор про написание драйверов режима ядра, необходимо узнать, каким образом драйверы регистрируются в системе и запускаются. Поэтому первым понятием, с которым мы столкнемся при работе с драйверами, будут службы.

Службы (services) – это процессы пользовательского режима, для запуска и функционирования которых регистрация интерактивного пользователя в системе не требуется. И поэтому подавляющее большинство служб не имеют пользовательского интерфейса. Это единственная категория пользовательских приложений, которые могут работать в таком режиме. Службы используются, например, для реализации серверов. Они могут запускаться как при загрузке операционной системы, так и после нее.

Однако термин «службы» применяется еще и в отношении драйверов устройств. Иными словами, в терминологии Microsoft по неясным причинам оказались объединены службы, выполняющиеся в режиме пользователя, и драйверы устройств, работающие в режиме ядра.

Для запуска и управления драйвером необходимы три компонента:

- диспетчер управления службами (Service Control Manager, SCM). Именно благодаря нему мы будем иметь возможность легко и просто загружать драйверы;
- программа управления службой (Service Control Program, SCP). Это обычная программа третьего кольца; она работает с диспетчером управления службами (вызывает функции, которые он предоставляет), чтобы установить, запустить драйвер и завершить его работу;
- собственно сам драйвер;
- для запуска службы (далее – драйвера) необходимо зарегистрировать драйвер. За этот процесс в системе отвечает функция `CreateService`. После регистрации драйвера в системе необходимо его запустить – за это отвечает функция `StartService`. Также при работе с драйвером могут понадобиться следующие функции: `ControlService`, `DeleteService`, `CloseServiceHandle`.

Кроме того, для возможности работы с диспетчером управления службами необходимо получить к нему доступ (`OpenSCManager`). Функция `CloseServiceHandle` также позволяет закрыть описатель, который возвращает функция `OpenSCManager`. Все функции, которые предоставляет диспетчер управления службами, находятся в библиотеке advapi32.dll.

Написание программы управления драйверами не входит в задачу данного раздела, поэтому писать программу мы не будем, а воспользуемся готовой бесплатной программой с открытым исходным кодом KmdManager (программа написана на MASM). Программу KmdManager можно найти на компакт-диске в папке utils.

На конечном этапе загрузки системы перед появлением диалога регистрации пользователя запускается SCM (\%SystemRoot%\System32\Services.exe), который, просматривая раздел реестра HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\, создает свою внутреннюю базу данных (ServicesActive database или SCM database). Далее SCM находит в созданной базе все драйверы устройств и службы, помеченные для автоматического запуска, и загружает их.

Чтобы получить кое-какое представление об этом, запустите редактор реестра, откройте раздел HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\ и изучите его содержимое.

3.3.2. Общий обзор

Итак, мы получили базовые сведения о службах в Windows. Теперь нам надо понять, что такое драйвер. Обычно под этим словом понимается драйвер устройства. Как следует из самого названия, *драйвер устройства* (device driver) – это программа, предназначенная для управления каким-то устройством, причем не обязательно физическим, а, возможно, и виртуальным. Именно о драйверах для управления такими виртуальными устройствами мы и будем вести речь в дальнейшем. Однако знания, полученные при написании виртуального устройства, вполне могут пригодиться при написании драйвера реального устройства. Цель данного раздела – предоставить базовые знания, необходимые для написания драйвером режима ядра для операционных систем Win32.

Существует несколько видов драйверов режима ядра (Kernel-Mode Drivers):

- драйверы файловых систем (File System Drivers) – реализуют ввод-вывод на локальные и сетевые диски;
- унаследованные драйверы (Legacy Drivers) – написаны для предыдущих версий Windows NT;
- драйверы видеоадаптеров (Video Drivers) – реализуют графические операции;
- драйверы потоковых устройств (Streaming Drivers) – реализуют ввод-вывод видео и звука;
- WDM-драйверы (Windows Driver Model, WDM) – поддерживают технологию Plug & Play и управления электропитанием. Их отличительной особенностью является совместимость на уровне исходного кода между Windows 98, Windows ME и Windows NT.

В разных источниках вы можете встретить классификацию, немного отличную от приведенной выше, но это не важно. Важно то, что драйверы, которые мы будем писать, не подпадают ни под один из пунктов этой классификации. Это не драйверы файловой системы, не унаследованные драйверы, не драйверы видеоадаптеров или звуковых карт и не WDM-драйверы, т. к. они не поддерживают Plug & Play и управление электропитанием. Де-факто это просто программы, работающие на нулевом уровне привилегий – система сама позволяет легко и просто добавить в неё код для непонятно какого устройства и делать с ней все что угодно!

Однако это не является дырой в системе безопасности – просто система так работает. Иначе и быть не может, поскольку, взаимодействуя с окружением, система

вынуждена предоставлять к себе доступ. В противном случае это была бы полностью закрытая, а значит, бесполезная система.

По своей структуре драйвер является не чем иным, как файлом PE-формата – таким же, как обычные EXE- и DLL-файлы. Только загружается и работает он по другим правилам. Драйверы можно рассматривать как DLL режима ядра, предназначенные для выполнения задач, не решаемых из третьего уровня привилегий. Принципиальная разница здесь (не считая уровня привилегий) в том, что мы не сможем напрямую обращаться ни к драйверу, ни к его коду, ни к его данным, а будем пользоваться специальным механизмом, предоставляемым диспетчером ввода-вывода (Input/Output Manager). Также драйвер не может обращаться к объектам пользовательской подсистемы (к объектам третьего кольца), а также создавать консоли, окна и графические объекты и работать с ними. Он может работать лишь с системными объектами, а именно: файлами, реестром, процессами, потоками и др. Диспетчер ввода-вывода обеспечивает среду для функционирования драйверов, а также предоставляет механизмы для их загрузки, выгрузки и управления ими.

3.3.3. Driver Development Kit (DDK)

Говоря про программирование драйверов в Windows, обязательно следует упомянуть Driver Development Kit (DDK). Что такое DDK? DDK – это набор инструментов, необходимых для создания драйверов для Windows.

К сожалению, Microsoft больше не распространяет DDK бесплатно, но при очень большом желании его все же можно найти. В этот пакет входит документация, которая является богатым, а иногда и единственным источником информации о внутренних структурах данных и внутрисистемных функциях, используемых драйверами устройств.

DDK предназначен для разработки драйверов на языке C. В этой книге речь идет о программировании на ассемблере. Программисту на ассемблере при программировании драйверов без DDK всё же не обойтись. Во-первых, понадобится документация, во-вторых – заголовочные файлы, которые содержат объявления констант и структур данных, используемых в ядре (правда, их придется переводить на синтаксис, понятный ассемблеру); ну и, в-третьих, конечно же, примеры исходных кодов драйверов (которые, разумеется, написаны на языке C).

В примерах драйверов используются заголовочные файлы из папки DDK. Папка DDK не поставляется вместе с компилятором FASM. Папка DDK, содержащая заголовочные файлы, необходимые для создания драйверов, содержится на компакт-диске, прилагающемся к данной книге. Заголовочные файлы в папке DDK не претендуют на звание самых полных – в них содержатся только те структуры данных, которые прямо или косвенно используются в примере к данному разделу.

Также помимо DDK, который предназначен для определённой версии Windows, компания Microsoft распространяет комплект WDK, который содержит инструменты и файлы, необходимые для разработки драйверов под все версии Windows.

3.3.4. Контекст потока и уровни запросов прерываний

В большинстве случаев количество приложений в несколько десятков раз больше, чем количество процессоров – и, естественно, для создания иллюзии одновременного их выполнения надо последовательно подключать эти приложения к процессору, причем очень быстро. Эта процедура называется *переключением контекста потока*. Если система переключает контекст потоков, принадлежащих одному процессу, то необходимо сохранить значение регистров процессора отключаемого потока, загрузить предварительно сохраненные значения регистров процессора подключаемого потока и обновить другие структуры данных. Если же подключаемый поток принадлежит другому процессу, то помимо этого необходимо загрузить в регистр CR3 процессора указатель на каталог страниц процесса. Т. к. каждому пользовательскому процессу предоставлено закрытое адресное пространство, то у разных процессов разные проекции адресных пространств, а значит, и разные каталоги страниц и наборы таблиц страниц, по которым процессор транслирует виртуальные адреса в физические.

Все это не имеет прямого отношения к программированию драйверов. Но напоминание о переключении контекстов связано вот с чем. Поскольку переключение контекста – операция не самая быстрая, то драйверы для достижения лучшей производительности, как правило, не создают своих потоков. Но код драйвера все же нужно выполнять. Поэтому для экономии времени на переключение контекстов драйверы выполняются в режиме ядра в одном из трех контекстов:

- в контексте пользовательского потока, инициировавшего запрос ввода-вывода;
- в контексте системного потока режима ядра (эти потоки принадлежат процессу System);
- как результат прерывания (а значит, в контексте потока, который был текущим на момент прерывания).

Если инициируется запрос ввода-вывода, то мы находимся в контексте потока, этот запрос инициировавшего, и значит, можем напрямую обращаться к адресному пространству процесса, которому этот поток принадлежит. Если же мы находимся в контексте системного потока, то ни к какому пользовательскому процессу обращаться напрямую не можем, а к системному и так всегда можем обратиться. Если, например, нужно посмотреть из драйвера, что расположено у такого-то процесса по такому-то адресу, то придется либо самим переключать контекст, либо по таблицам страниц транслировать адреса.

Прерывание – неотъемлемая часть любой операционной системы. Прерывание требует обработки, поэтому выполнение текущего кода прекращается и управление передается обработчику прерывания. Существуют как аппаратные, так и программные прерывания. Прерывания обслуживаются в соответствии с их приоритетом. Windows NT использует схему приоритетов прерываний, известную под названием «уровни запросов прерываний» (interrupt request levels, IRQL). Всего существует 32 уровня – начиная с 0 (passive), имеющего самый низкий приоритет, и до 31 (high) с самым высоким приоритетом. Причем прерывания с IRQL=0 по IRQL=2 (DPC\dispatch) являются программными, а прерывания с IRQL=3

(device 1) по IRQL=31 (high) – аппаратными. Не путайте уровни приоритета прерываний с уровнями приоритетов потоков (это совсем разные вещи!), а также с уровнями приоритета аппаратных прерываний, которые используются контроллером прерываний. Прерывание с уровнем IRQL=0, строго говоря, прерыванием не является, т. к. оно не может прервать работу никакого кода (ведь для этого код должен выполняться на еще более низком уровне прерывания, которого не существует). На этом IRQL выполняются потоки пользовательского режима.

Существуют функции ядра, позволяющие узнать текущий уровень прерывания, а также повысить или понизить его. На уровне прерывания passive можно вызывать любые функции ядра (в DDK в описании каждой функции обязательно указано, на каком уровне прерывания ее можно вызывать), а также обращаться к страницам памяти, сброшенным в файл подкачки. На более высоких уровнях прерывания (DPC/dispath и выше) попытка обращения к странице, отсутствующей в физической памяти, приводит к краху системы, т. к. диспетчер памяти не может обрабатывать ошибки страниц.

3.3.5. Пример простого драйвера

Пришло время познакомиться с драйверами поближе. Итак, посмотрим код простейшего драйвера. Драйвер, код которого приведён в листинге 3.19, после загрузки генерирует звуковой сигнал и выгружается из памяти.

Листинг 3.19. Пример простейшего драйвера

```
format PE Native 4.0

include 'win32w.inc'
include 'DDK\ntstatus.inc'

entry DriverEntry

TIMER_FREQUENCY     equ 1193167
OCTAVE         equ 2

DO      equ 523         ; До    -  523,25 Гц
MI      equ 659         ; Ми    -  659,25 Гц
SOL       equ 784       ; Соль -  783,99 Гц

TONE1       equ TIMER_FREQUENCY/(DO*OCTAVE)
TONE2       equ TIMER_FREQUENCY/(MI*OCTAVE)
TONE3       equ TIMER_FREQUENCY/(SOL*OCTAVE)

section '.code' code readable executable

proc Sound dwPitch

   cli

   invoke WRITE_PORT_UCHAR, 43h, 10110110b
```

```
   mov eax, [dwPitch]
   out 42h, al

   mov al, ah
   out 42h, al

   ; включить динамик

   invoke READ_PORT_UCHAR, 61h
   or al, 11b
   invoke WRITE_PORT_UCHAR, 61h, eax

   sti

   mov ecx, 5000000h
   loop $

   cli

   ; выключить динамик

   invoke READ_PORT_UCHAR, 61h
   and al, 11111100b
   invoke WRITE_PORT_UCHAR, 61h, eax

   sti

   ret

endp

proc DriverEntry DriverObject, RegistryPath

   stdcall  Sound, TONE1
   stdcall  Sound, TONE2
   stdcall  Sound, TONE3

   mov eax, STATUS_DEVICE_CONFIGURATION_ERROR
   ret

endp

section '.relocs' fixups readable writeable discardable

section '.idata' import readable writeable

  library hal,'hal.dll'

  import hal,\
    WRITE_PORT_UCHAR, 'WRITE_PORT_UCHAR',\
    READ_PORT_UCHAR, 'READ_PORT_UCHAR'
```

Итак, первая особенность, которая сразу бросается в глаза, – это первая строка `format PE Native 4.0` – она говорит компилятору, чтобы он генерировал файл драйвера с расширением SYS.

Начнём по порядку. Как и в любой программе для третьего кольца, мы включаем файл win32w.inc, содержащий стандартные объявления, необходимые для программирования в Win32, и файл ntstatus.inc, содержащий коды ошибок, используемые в ядре Windows. Потом опять же традиционно мы указываем точку входа – после загрузки драйвера в память система передаст управление коду, расположенному в точке входа драйвера. Далее идут объявления констант, задающих частоты звуков, которые будут раздаваться из системного динамика.

После констант идёт секция кода, где есть две функции – `Sound` и `DriverEntry`. Рассмотрим сначала функцию `DriverEntry`. Как и у любого другого выполнимого модуля, у драйвера должна быть точка входа, на которую система передаст управление после загрузки драйвера в память. У нас, как и в текстах на языке C, это `DriverEntry`, которая оформлена в виде процедуры, принимающей два параметра. Имя процедуры, естественно, может быть любым, но чтобы не отходить от общепринятой номенклатуры, назовём ее так, как её называет Microsoft.

Итак, функция `DriverEntry` принимает два параметра: `DriverObject` и `RegistryPath`. Что они обозначают?

1. `DriverObject` – указатель на объект только что созданного драйвера. Windows является объектно-ориентированной системой. При этом понятие «объект» распространяется на все что только можно, и драйверы не являются исключением. Загружая драйвер, система создает объект-драйвер, представляющий для нее образ драйвера в памяти. Через этот объект система управляет драйвером. Звучит красиво, но не дает никакого представления о том, что же в действительности происходит! Если отбросить всю эту объектно-ориентированную «мишуру», то станет очевидно, что объект-драйвер представляет собой обыкновенную структуру данных типа `DRIVER_OBJECT` (определена в ntddk.inc). Некоторые поля этой структуры заполняет система, а некоторые придется заполнять нам самим. Обращаясь к этой структуре, система и управляет драйвером.
   Итак, как вы наверное уже поняли, первым параметром, передающимся в функцию `DriverEntry`, как раз и является указатель на эту самую структуру (или, пользуясь объектно-ориентированной терминологией, – объект-драйвер). Используя этот указатель, мы можем (и будем, но позже) заполнять соответствующие поля структуры `DRIVER_OBJECT`. Но в рассматриваемом драйвере это не требуется.
2. `RegistryPath` – указатель на раздел реестра, содержащий параметры инициализации драйвера. Точнее говоря, это указатель на структуру типа `UNICODE_STRING`. А уже в ней содержится указатель на саму UNICODE-строку, содержащую имя раздела. Структура `UNICODE_STRING` будет пояснена чуть позже.

Получив управление, драйвер генерирует три звуковых сигнала и возвращает управление. Функция `Sound` принимает один параметр: значение коэффициента

пересчёта для системного таймера. Это значение нужно для того, чтобы динамик издавал нужный звук. Не стоит вдаваться в подробности работы этой процедуры – она всего лишь демонстрирует возможности драйвера, а именно – его способность работать с оборудованием напрямую через порты ввода/вывода, как с использованием инструкций `in/out`, так с использованием функций из библиотеки hal.dll.

После выполнения действий в процедуре `DriverEntry` надо вернуть системе некое значение, указывающее на то, как прошла инициализация драйвера. Если вернуть `STATUS_SUCCESS`, то инициализация считается успешной и драйвер остаётся в памяти. Любое другое значение `STATUS_*` указывает на ошибку, и в этом случае драйвер выгружается системой. Вышеприведенный драйвер является простым: единственное, что он делает, – позволяет себя загрузить и генерирует через системный динамик три звуковых сигнала. Поскольку ничего кроме этого он сделать больше не может, то возвращает код ошибки `STATUS_DEVICE_CONFIGURATION_ERROR` (полный список можно посмотреть в файле ntstatus.inc). Если возвратить `STATUS_SUCCESS`, то драйвер так и останется в памяти безо всякой пользы, и выгрузить его средствами SCM будет невозможно, т. к. мы не определили процедуру, отвечающую за выгрузку драйвера. Если загружать драйвер при помощи программы KmdManager, то после загрузки драйвера и подачи звукового сигнала в колонке `LastError` будет написана ошибка «Параметр задан неверно». Эта ошибка режима пользователя наиболее близка ошибке `STATUS_DEVICE_CONFIGURATION_ERROR` режима ядра.

После секции кода идёт секция релокейшенов. Как и для DLL, для драйвера секция релокейшенов является жизненно необходимой, потому что неизвестно, по какому адресу драйвер будет загружен, а секция релокейшенов содержит информацию о том, какие инструкции надо подправить, если загрузка прошла не по тому адресу. Следует обратить внимание, что эта секция помечена как `discardable`. Флаг `discardable` говорит менеджеру памяти, что эта секция может быть удалена из памяти, в случае если выявится нехватка памяти.

Последней идёт секция с таблицей импорта. Таблица импорта ничем не отличается от таблицы импорта программ, работающих на третьем уровне привилегий: мы указываем модули, которые хотим использовать, и функции из этих модулей.

3.3.6. Строки в ядре Windows

В разделе 3.3.5 мы оставили без внимания структуру `UNICODE_STRING`. Работа со строками в ядре Windows отличается от работы со строками в третьем кольце.

Если в третьем кольце функция принимала строку в качестве параметра, то у неё было два варианта: вариант, работающий с ANSI-строкой, и вариант, работающий с UNICODE-строкой. Также если функция принимала строку в качестве параметра, то надо было просто передать указатель на её первый символ – конец строки определялся по нулевому символу.

В нулевом кольце система работает только с UNICODE–строками, и, чтобы передать строку, недостаточно передать указатель на её первый символ. В режиме ядра строка описывается специальной структурой под названием `UNICODE_STRING`.

```
struc UNICODE_STRING
{
   .Length         dw ?
```

```
    .MaximumLength  dw ?
    .Buffer         dd ?
}
```

Поле `Length` содержит текущую длину строки в байтах (не в символах!), не считая завершающих двух нулей. Поле `MaximumLength` содержит максимальный размер буфера (также в байтах), в котором эта строка присутствует. Поле `Buffer` содержит указатель на буфер, где находится искомая строка. Как видно, структура `UNICODE_STRING` содержит поле, содержащее длину строки, что избавляет функцию, принимающую строку, от некоторых лишних вычислений. Также в структуре `UNICODE_STRING` присутствует поле, содержащее максимальный размер буфера, что избавляет программу от передачи дополнительных параметров функциям.

Для удобства работы со строками ядро Windows предоставляет целый набор функций, упрощающих работу с таким форматом строк, например: `RtlInitUnicodeString`, `RtlUnicodeStringToAnsiString`, `RtlUpcaseUnicodeString`, `RtlAnsiStringToUnicodeString`, `RtlInitAnsiString`, `RtlCompareUnicodeString` и т. д.

3.3.7. Подсистема ввода-вывода

Подсистема ввода-вывода является одним из главных аспектов при программировании драйверов режима ядра. В начале раздела 3.3 было сказано, что разрабатываемые нами драйверы можно считать DLL режима ядра. С определенной долей условности это действительно так. Зачем еще нужен драйвер, который не управляет каким-либо реальным устройством? Только для того, чтобы служить проводником в режим ядра. При этом код драйвера, по сути, является набором функций, позволяющих решать задачи, недоступные коду режима пользователя. Когда нужно решить одну из таких задач, вызывается соответствующая функция. Принципиальная разница (не считая уровня привилегий) заключается в том, что в случае с обычной DLL мы получаем (явно или неявно) адрес интересующей нас функции и передаем туда управление. В случае с драйвером режима ядра такой сценарий был бы крайне рискованным с точки зрения безопасности системы. Поэтому система предоставляет посредника в лице диспетчера ввода-вывода (I/O manager), который является одним из компонентов подсистемы ввода-вывода. Диспетчер ввода-вывода отвечает за формирование пакета запроса ввода-вывода (I/O request packet, IRP) и отправку его драйверу для дальнейшей обработки.

На рис. 3.3 представлена общая схема работы подсистемы ввода-вывода в системах WinNT. Здесь изображены наиболее важные объекты, которые должны фигурировать в процессе ввода-вывода, а именно:

- объект-драйвер;
- объект-устройство;
- объект-файл;
- структура IRP.

Также в случае взаимодействия кода третьего кольца с кодом нулевого кольца обязательно должна присутствовать символьная ссылка.

В образном смысле объект-драйвер можно уподобить адресату, которому присылают письма с запросами. Объект-устройство – это «почтовый ящик», на который присылают письма. Структура IRP – это само «письмо»: она-то как раз и являет

собой запрос ввода-вывода, т. е. в ней содержится вся информация о запросе. Объект-файл – это «почтовая марка»: без неё запрос не будет послан – она разрешает отправку. На один и тот же объект-устройство можно получить объект-файл только для записи или только для чтения – в зависимости от того, какие параметры будут указаны при открытии объекта-устройства (функции `CreateFile` или `NtCreateFile`). Например, драйвер может давать доступ на запись в устройство только определённому списку доверенных приложений.

Итак, давайте посмотрим, что должен сделать драйвер для нормального взаимодействия с кодом третьего кольца. Он должен создать два объекта: объект-устройство и объект «символьная ссылка»; также драйвер должен определить функции-обработчики пакетов запроса ввода-вывода.

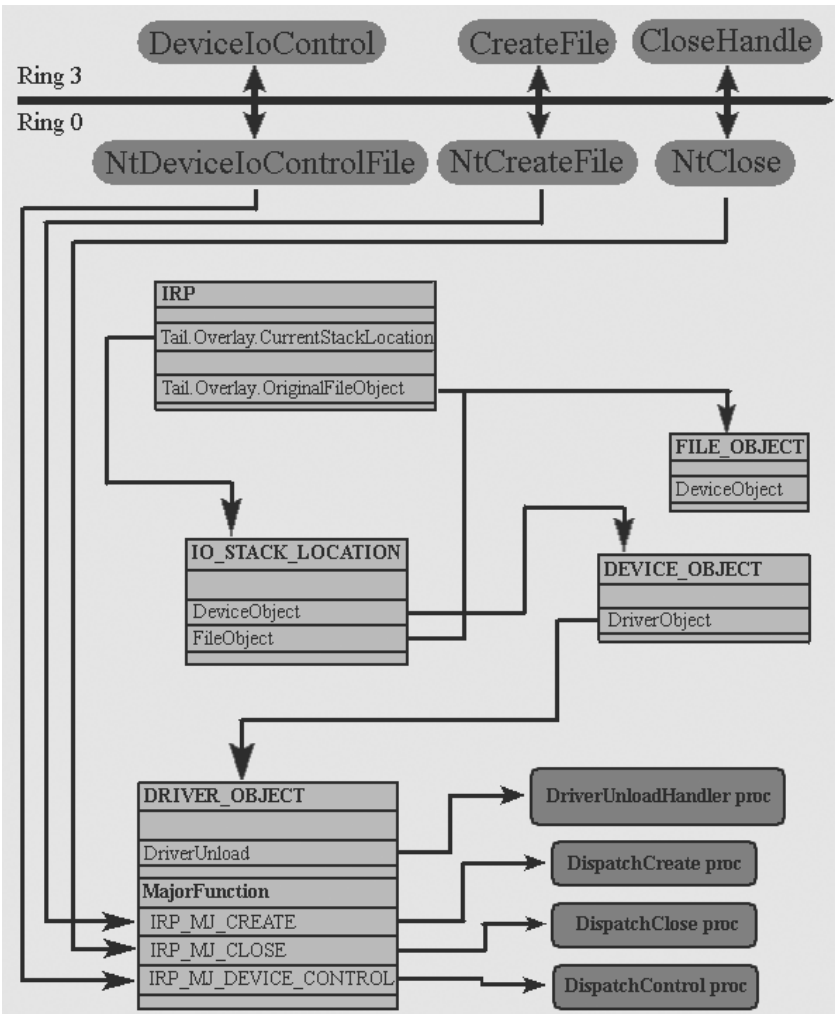


Рис. 3.3. Общая схема работа подсистемы ввода-вывода

В первую очередь драйвер должен создать объект-устройство – без этого взаимодействие с кодом режима пользователя при помощи диспетчера ввода вывода будет невозможно. Как у большинства объектов, у объекта-устройства должно быть имя. За создание объекта-устройства отвечает функция `IoCreateDevice`, которую экспортирует модуль ntoskrnl.exe.

Поскольку создается некий новый объект с определенным именем, на сцену выходит еще один ключевой компонент системы, который мы до этого момента еще не упоминали – диспетчер объектов. Он отвечает за создание, защиту объектов и управление ими. Имена объектов попадают в пространство имен диспетчера объектов. Пространство имен имеет иерархическую структуру, аналогичную структуре каталогов файловой системы. Для просмотра базы данных диспетчера объектов можно использовать разные утилиты (например, WinObj Sysinternals, доступную на сайте sysinternals.com).

Все каталоги в пространстве имен диспетчера объектов, кроме двух – \BaseNamedObjects и \?? (GLOBAL??), невидимы для кода режима пользователя. Поэтому ни к одному объекту, за исключением находящихся в этих двух каталогах, код режима пользователя обратиться не может. Это сделано все по тем же соображениям безопасности. Таким образом, ни к объекту-драйверу в каталоге \Driver, ни к объекту-устройству в каталоге \Device код режима пользователя не обращается.

Для того чтобы объект-устройство стал доступен коду режима пользователя, драйвер должен создать в каталоге \??, доступном этому коду, еще один объект – символьную ссылку. Такое странное имя этот каталог получил потому, что при сортировке по алфавиту он будет первым, что увеличивает скорость поиска объектов. До Windows NT4 этот каталог назывался \DosDevices и являлся наиболее часто используемым при поиске объектов – потому и был переименован в \??. Для совместимости с драйверами предыдущих версий Windows в корневом каталоге пространства имен диспетчера объектов имеется символьная ссылка \DosDevices, значением которой является строка \?? (рис. 3.4). За создание объекта «символьная ссылка» отвечает функция `IoCreateSymbolicLink`, которую экспортирует модуль ntoskrnl.exe.

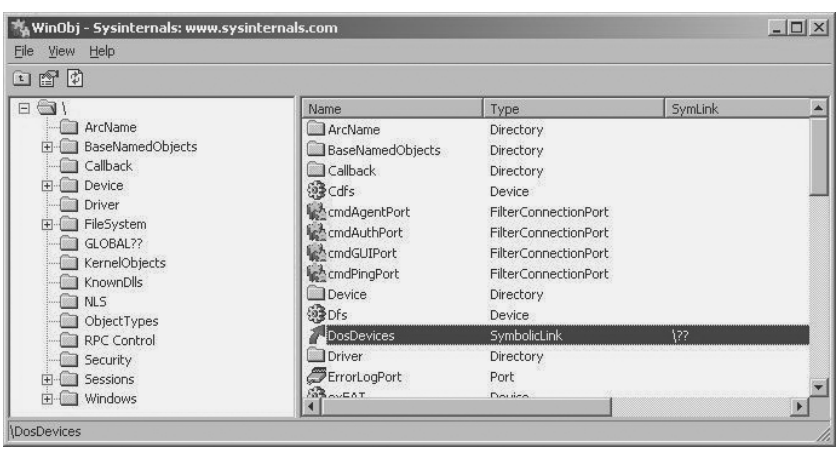


Рис. 3.4. Пространство имён диспетчера объектов

Итак, объекты созданы – теперь необходимо определить функции-обработчики запросов к устройству. Для возможности взаимодействия с созданным устройством драйвер должен определить обработчики как минимум трёх следующих запросов ввода-вывода: `IRP_MJ_CREATE`, `IRP_MJ_CLOSE` и `IRP_MJ_DEVICE_CONTROL`. Запрос `IRP_MJ_CREATE` генерируется при вызове кодом пользователя функции `CreateFile` либо функцией `NtCreateFile` в ядре. Запрос `IRP_MJ_CLOSE` генерируется при закрытии хендла объекта (функции `CloseHandle` и `NtClose`). Запрос `IRP_MJ_DEVICE_CONTROL` генерируется при вызове функции `DeviceIoControl` в режиме пользователя либо `NtDeviceIoControlFile` в режиме ядра.

Обслуживание запросов ввода-вывода в большинстве случаев происходит при обработке запросов `IRP_MJ_DEVICE_CONTROL`, вернее в функции-обработчике запроса `IRP_MJ_DEVICE_CONTROL`. Но как же драйвер узнает, что именно от него хотят? Необходимо как-то дифференцировать запросы. Для этого используется управляющий код ввода-вывода, который строится по определённым правилам. На рис. 3.5 приведён его формат.

`DeviceType` – идентификатор типа устройства (16 бит). Может принимать значение в диапазоне 0–0FFFFh, который разбит на две равные половины. Диапазон 0–7FFFh зарезервирован Microsoft, а диапазон 8000h–0FFFFh доступен всем желающим. В файле ntddk.inc можно найти набор констант `FILE_DEVICE_*` со значениями из зарезервированного диапазона. Мы будем использовать `FILE_DEVICE_UNKNOWN`. Можно определить и свой собственный идентификатор.

`Access` – запрашиваемые права доступа к устройству. Поскольку это поле размером 2 бита, то возможны четыре значения:

- `FILE_ANY_ACCESS` (0) – максимальные права доступа;
- `FILE_READ_ACCESS` (1) – доступ на чтение, т. е. получение данных от устройства;
- `FILE_WRITE_ACCESS` (2) – доступ на запись, т. е. передачу данных на устройство;
- `FILE_READ_ACCESS or FILE_WRITE_ACCESS` (3) – комбинация последних двух значений.

`Function` – собственно и определяет операцию, которую должно выполнить устройство (12 бит). Может принимать значение в диапазоне 0–FFFh, который также разбит на две равные половины. Диапазон 0–7FFh зарезервирован Microsoft, а диапазон 800h–0FFFh доступен.

`Method` – определяет метод ввода-вывода. Размер поля 2 бита – следовательно, возможны четыре значения, и все они могут быть использованы для наших целей:

`METHOD_BUFFERED` (0) – буферизованный ввод-вывод;

`METHOD_IN_DIRECT` (1), `METHOD_OUT_DIRECT` (2) – прямой ввод-вывод;

`METHOD_NEITHER` (3) – ввод-вывод без управления.

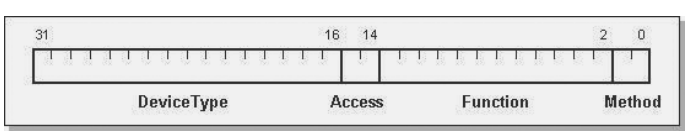


Рис. 3.5. Формат управляющего кода ввода-вывода

При буферизированном вводе-выводе диспетчер ввода-вывода выделяет в системном пуле неподкачиваемой памяти буфер, равный по размеру наибольшему из пользовательских буферов (если определен как входной, так и выходной).

Создавая IRP при операции записи, диспетчер ввода-вывода копирует данные из пользовательского входного буфера в выделенный системный буфер и передает драйверу его адрес в поле `AssociatedIrp.SystemBuffer` структуры `_IRP`. Размер скопированных данных помещается в поле `Parameters.DeviceIoControl.InputBufferLength` структуры `IO_STACK_LOCATION` (на эту структуру указывает поле `Tail.Overlay.CurrentStackLocation` структуры `_IRP`).

Драйвер обрабатывает IRP и помещает выходные данные (при наличии таковых) в тот же самый системный буфер. При этом находящиеся там входные данные, естественно, переписываются. Т. е. пользовательских буферов два, а системный один. Если входные данные нужны драйверу для дальнейшей работы, то он должен их где-нибудь сохранить.

Завершая IRP при операции чтения, диспетчер ввода-вывода копирует данные из системного буфера в пользовательский выходной буфер. Размер копируемых данных извлекается из поля `IoStatus.Information` структуры _IRP. Это поле должен заполнить драйвер – только он «знает», сколько байтов поместил в системный буфер.

Таким образом, диспетчер ввода-вывода дважды копирует буферы. Именно поэтому данный вид управления – самый медленный, но при сравнительно малых размерах буферов издержки незначительны.

У этого вида управления есть одно большое преимущество: диспетчер ввода-вывода сам решает все проблемы, связанные с возможными ошибками доступа к памяти. В худшем случае вызов функции `DeviceIoControl` завершится неудачно на стороне режима пользователя.

Следовательно, этот вид управления буферами стоит использовать при относительно небольших размерах буферов или в случаях, если операция ввода-вывода выполняется не часто. Если же драйверу приходится перегонять большие порции данных, например потоки аудио- или видеоданных (здесь мы даже вскользь не коснемся этой темы), то необходимо использовать более эффективный метод.

Прямой ввод-вывод используется для организации прямого доступа к памяти (direct memory access, DMA). Данный вид управления не будет рассматриваться в этом разделе. Коротко говоря, ситуация с входным пользовательским буфером полностью аналогична предыдущему виду управления. Выходной же буфер, несмотря на свое название, может быть использован как источник (`METHOD_IN_DIRECT`) или приемник (`METHOD_OUT_DIRECT`) данных.

Диспетчер ввода-вывода блокирует выходной буфер в памяти, делая его неподкачиваемым, и создает список MDL (Memory Descriptor List), описывающий физические страницы, занимаемые буфером. Указатель на MDL помещается в поле `MdlAddress` структуры _IRP. Получив для данного буфера описание в виде MDL, драйвер в дальнейшем может использовать буфер в контексте памяти любого процесса.

Данный вид управления подходит для передачи больших порций данных. В данной книге этот способ рассматриваться не будет.

При использовании ввода-вывода без управления диспетчер ввода-вывода помещает в поле `DeviceIoControl.Type3InputBuffer` структуры `IO_STACK_LOCATION` указатель на пользовательский входной буфер, а в поле `UserBuffer` структуры `_IRP` – указатель на пользовательский выходной буфер и оставляет драйверу возможность управлять ими самостоятельно. Таким образом, вся ответственность за управление пользовательскими буферами ложится на драйвер. Он может блокировать их в памяти, отображать на системное адресное пространство или обращаться к ним напрямую и т. д.

Остается одна проблема – пользовательский поток может передать заведомо неверный адрес буфера, например попадающий в системное адресное пространство или адрес невыделенной области памяти и т. п. Или пользовательский процесс может быть многопоточным, и один из потоков может освободить память, занимаемую буфером во время обработки драйвером запроса ввода-вывода.

Ввод-вывод без управления является самым небезопасным методом передачи данных, т. к. может привести к непредвиденным ошибкам при работе с буфером. Любая необработанная ошибка в режиме ядра приводит к «синему экрану смерти» (BSOD); также следует помнить, что некоторые ошибки, возникшие в ядре, приводят к BSOD в любом случае. Такой метод управления был бы полезен при обработке запросов от других драйверов. Тем не менее использовать его следует с большой осторожностью.

Итак, что происходит при вызове функции `DeviceIoControl`? Сначала она подготавливает параметры для функции `NtDeviceIoControlFile`, которая является её заменителем в ядре, и переходит в нулевое кольцо, вызвав системный сервис командой `int 2Eh` (win2k) или командой `sysenter` (Windows XP и старше). В нулевом кольце функция `NtDeviceIoControlFile` передаёт управление диспетчеру ввода-вывода. Диспетчер в свою очередь копирует входной буфер в системную память (если это предполагает выбранный режим управлением буфером), формирует пакет запроса IRP и вызывает функцию-обработчик запроса `IRP_MJ_DEVICE_CONTROL`. Функция-обработчик выполняет все требуемые от неё действия и возвращает управление диспетчеру ввода-вывода. Последний, проанализировав результаты работы функции-обработчика запроса, копирует данные из системного буфера в пользовательский выходной буфер (разумеется, если это требуется), возвращает управление функции `NtDeviceIoControlFile`, а та в свою очередь возвращает управление функции `DeviceIoControl`, если вызов произошёл из третьего кольца.

Стоит заметить, что вызов функции `DeviceIoControl` – не единственный случай, который приводит к вызову функций драйвера: также этот вызов происходит при обращении к функциям `ReadFile`, `WriteFile` и т. д. (разумеется, если в драйвере определён их обработчик). Например, при обращении к функции `ReadFile` формируется запрос типа `IRP_MJ_READ`.

Операционная система абстрагирует запросы ввода-вывода, обрабатывая их так, будто они адресованы файлам. Драйвер преобразует запросы к виртуальному файлу в запросы, специфичные для устройства, скрывая тот факт, что конечное

устройство может и не быть устройством с файловой структурой. Такая запутанная схема позволяет унифицировать интерфейс приложений и устройств. Т. е. все считываемые или записываемые данные представляются потоками байтов, направляемыми в виртуальные файлы.

Следует заметить, что инициатором ввода-вывода является код не только третьего, но и нулевого кольца. Ничто не мешает другим драйверам в системе обращаться к созданному нашим драйвером устройству.

3.3.8. Практика

Итак, базовые сведения о работе драйверов и о подсистеме ввода-вывода мы рассмотрели – теперь необходимо изучить работу драйвера более подробно на конкретном примере. Напишем драйвер, который передает процессу третьего кольца содержимое глобальной дескрипторной таблицы. Полный исходный код программы управления драйвером и исходный код самого драйвера можно найти на компакт-диске в папке part3.

3.3.8.1. Программа управления драйвером

Программа управления драйвером будет загружать драйвер, осуществлять запрос к нему и выгружать его из памяти. В листинге 3.20 представлен код приложения, осуществляющего управление драйвером (приведены наиболее важные моменты).

Листинг 3.20. Программа управления драйвером

```
format PE GUI 4.0

...
entry start

IOCTL_DEFINE_EX IOCTL_DUMP_GDT, FILE_ANY_ACCESS, 1, METHOD_BUFFERED

section '.data' data readable writeable

DriverFileName db 'E:\FASM\PROJECTS\drivers\gdtdump.sys',0
DriverServiceName db 'gdtdump',0
DeviceName db '\\.\GDTDump',0
DriverDisplayName db 'GDT dumper',0
...

inputData:
   .Offset dd ?
   .Size  dd ?

FS_Descriptor dq ?
DescriptorInfo:
   db 'Base address : '
.BaseAddr db '   '
          db 13, 10
   db 'Limit : '
```

```
.Limit    db '   '
          db 13, 10
   db 'DPL : '
.DPL   db ' ',0


align 10h
Buffer db 10000h dup (?)
BufferSize = $-Buffer

section '.code' code readable executable

...

LoadDriver:
     invoke, 0, 0, SC_MANAGER_ALL_ACCESS
     cmp eax, 0
     jz @f
     mov [hSCManager], eax

     invoke CreateService, eax, DriverServiceName, DriverDisplayName,
SERVICE_START + SERVICE_STOP + DELETE, SERVICE_KERNEL_DRIVER, SERVICE_
DEMAND_START, SERVICE_ERROR_IGNORE, DriverFileName, 0, 0, 0, 0, 0
     cmp eax, 0
     jz @f
     mov [hService], eax

     invoke StartService, [hService], 0, 0
      @@:
     ret

UnLoadDriver:
     invoke ControlService, [hService], SERVICE_CONTROL_STOP, temp
     invoke DeleteService, [hService]
     invoke CloseServiceHandle, [hService]
     invoke CloseServiceHandle, [hSCManager]

     ret

start:

          call LoadDriver
          cmp eax, 0
       jnz @f

     invoke MessageBox,0,MsgFail,DriverDisplayName,0
     jmp .exit
      @@:
     invoke MessageBox,0,MsgSuccess,DriverDisplayName,0
```

```
    invoke CreateFile, DeviceName, GENERIC_READ + GENERIC_WRITE, FILE_
SHARE_READ, 0, OPEN_EXISTING, 0, 0
    mov [hDevice], eax

    invoke DeviceIoControl, eax, IOCTL_DUMP_GDT, 0, 0, Buffer, BufferSize,
Readed, 0


    invoke CreateFile, DumpFileName, GENERIC_WRITE, FILE_SHARE_READ, 0,
CREATE_ALWAYS, 0, 0
    push eax
    invoke WriteFile, eax, Buffer, [Readed], Readed, 0
    pop eax
    invoke CloseHandle, eax

    xor eax, eax
    mov ax, fs
    and al, 11111000b
    mov [inputData.Offset], eax
    mov [inputData.Size], 8
    invoke DeviceIoControl, [hDevice], IOCTL_DUMP_GDT, inputData, 8, FS_
Descriptor, 8, Readed, 0

    mov esi, FS_Descriptor
    call DecodeDescriptor

    push eax
    push ecx

    mov eax, ebx
    mov ebx, 16
    mov esi, DescriptorInfo.BaseAddr
    call dword_to_STR

    pop eax
    mov ebx, 16
    mov esi, DescriptorInfo.Limit
    call dword_to_STR

    pop eax
    mov ebx, 16
    mov esi, DescriptorInfo.DPL
    call dword_to_STR

    invoke MessageBox, 0, DescriptorInfo, DriverDisplayName, 0

    invoke CloseHandle, [hDevice]

  .exit:
```

```
call UnLoadDriver
push 0
call [ExitProcess]
```

Для работы с драйвером в первую очередь необходимо его загрузить. За это отвечает функция LoadDriver; вдаваться в подробности её работы мы не будем. Вкратце можно описать эту работу так: сначала при помощи функции OpenSCManager мы получаем хендл менеджера управления сервисами для возможности дальнейшей работы с ними, потом при помощи функции CreateService создаётся служба-драйвер, а после этого драйвер загружается в память (служба запускается) с помощью функции StartService. В случае неудачи функция LoadDriver возвращает в регистр EAX нулевое значение. После успешной загрузки драйвера в память выводится соответствующее сообщение.

Следующим шагом будет открытие устройства, созданного драйвером, для последующей работы с ним. Для возможности работы с устройством необходимо создать объект-файл. Это осуществляет функция CreateFile; в качестве имени файла надо передать ей имя устройства, созданного драйвером: \\.\GDTDump. Функция CreateFile подготовит параметры для своей функции-заменителя в ядре NtCreateFile и, выполнив команду int 2Eh или SYSENTER, переведёт выполнение программы в режим ядра.

NtCreateFile заменяет псевдоним локального компьютера \\.\ на имя каталога \?? в пространстве имен диспетчера объектов (таким образом, \\.\GDTDump превращается в \??\GDTDump) и вызывает функцию ядра ObOpenObjectByName. Через символьную ссылку ObOpenObjectByName функция NtCreateFile находит объект \Device\devGDTDump и возвращает указатель на него (таким образом, символьная ссылка, видимая коду режима пользователя, используется диспетчером объектов для трансляции во внутреннее имя устройства). Используя этот указатель, NtCreateFile создает новый объект-файл, представляющий устройство, и возвращает его описатель.

Прежде чем функция CreateFile вернет управление, в драйвере будет вызвана функция, которую определяет сам драйвер. Диспетчер ввода-вывода сформирует IRP типа IRP_MJ_CREATE и направит его драйверу, обслуживающему устройство. При этом процедура, обрабатывающая IRP_MJ_CREATE, будет выполнена в контексте потока, вызвавшего CreateFile при IRQL = PASSIVE_LEVEL. Если процедура вернёт код успеха, то новый описатель будет создан и возвращён коду, вызвавшему CreateFile. Если драйвер вернет код ошибки, то, соответственно, никаких новых описателей не возникнет.

Таким образом, описатель объекта «файл» и символьная ссылка на объект-устройство являются косвенными указателями на системные ресурсы, что позволяет системе оградить прикладные программы от прямого взаимодействия с системными структурами данных.

После успешного открытия устройства можно работать с этим ним, используя функцию DeviceIoControl. Она принимает восемь параметров:

1. Описатель устройства (HANDLE).
2. Управляющий код ввода вывода (DWORD).

3. Указатель на входной буфер (LPVOID).
4. Размер входного буфера (DWORD).
5. Указатель на выходной буфер (LPVOID).
6. Размер выходного буфера (DWORD).
7. Указатель на переменную, в которую будет сохранено количество байтов, которое было фактически скопировано в выходной буфер (LPDWORD).
8. Указатель на структуру OVERLAPPED, необходимую для асинхронных операций.

В принципе, здесь нет ничего сложного – кроме последнего параметра, который необходим при выполнении асинхронных операций. Поскольку в данном разделе и в данном примере асинхронный ввод-вывод не рассматривается, этот параметр мы тоже рассматривать не будем.

Первый вызов DeviceIoControl происходит для получения всей глобальной дескрипторной таблицы с последующим сохранением её в файл. Драйвер спроектирован, так что если входной буфер не указан, то он будет копировать в выходной буфер всю GDT; если же входной буфер указан, первый DWORD-элемент в нём будет указывать драйверу, с какой позиции в GDT копировать данные, а второе DWORD-значение – количество байтов.

Последующий вызов DeviceIoControl осуществляется в целях получения содержимого дескриптора, селектор которого указан в регистре FS. После получения дескриптора происходит вызов подпрограммы DecodeDescriptor; передав ей указатель на дескриптор, получаем базу, лимит и DPL дескриптора (они будут находиться в EBX, ECX и EAX соответственно). После получения базы, лимита и DPL дескриптора происходит преобразование их в строки и выводится сообщение.

После выполнения всех действий происходит выгрузка драйвера из памяти и удаляется служба, представляющая этот драйвер.

3.3.8.2. Драйвер GDTDump

Программа, работающая с драйвером, написана – теперь осталось написать сам драйвер. Вообще-то на практике всё делается наоборот: сначала драйвер, потом программа; тем не менее такой порядок изложения материала будет более понятен начинающим.

Далее в листинге 3.21 приведён полный исходный код драйвера GDTDump.

Листинг 3.21. Драйвер GDTDump

```
format PE Native 4.0

include 'win32w.inc'
include 'DDK\ntstatus.inc'
include 'DDK\ntdefs.inc'
include 'DDK\ntddk.inc'
include 'DDK\macros.inc'
include 'DDK\advmacro.inc'

entry DriverEntry
```

```
section '.code' code readable writeable executable

IOCTL_DEFINE_EX IOCTL_DUMP_GDT, FILE_ANY_ACCESS, 1, METHOD_BUFFERED

UNICODE_STRING_define DeviceName, "\Device\devGDTDump"
UNICODE_STRING_define SymbolicLinkName, "\??\GDTDump"

DeviceObject dd 0

GDTR:
  .Limit  dw ?
  .Address  dd ?

proc DispatchControl pDeviceObject, pIrp

local status:DWORD

    push edi
    push esi
    push ecx
    push edx

    mov [status], STATUS_DEVICE_CONFIGURATION_ERROR

    mov edi, [pIrp]
    virtual at edi
     .edi IRP
    end virtual

    mov esi, [.edi.Tail.Overlay.CurrentStackLocation]
    virtual at esi
     .esi IO_STACK_LOCATION
    end virtual

    cmp [.esi.Parameters.DeviceIoControl.IoControlCode], IOCTL_DUMP_GDT
    jz .IOCTL_DUMP
    jmp .next
   .IOCTL_DUMP:
    ;int3
    sgdt fword [GDTR]
    xor ecx, ecx
    mov cx, [GDTR.Limit]
    inc ecx  ; ecx = GDT size

    xor eax, eax
    mov edx, [.esi.Parameters.DeviceIoControl.InputBufferLength]
    cmp edx, 8
    jnz .copydata
    mov eax, [.edi.AssociatedIrp.SystemBuffer]
```

```
 mov edx, [eax+4]  ; edx = needed data size
 mov eax, [eax]    ; eax = start offset in GDT

 add edx, eax
 cmp edx, ecx
 jna @f
 mov edx, ecx
@@:
 sub edx, eax   ; edx = fact data size
 cmp edx, 7FFFFFFFh
 jna @f
 mov edx, 0
@@:
 mov ecx, edx
.copydata:
 mov [status], STATUS_BUFFER_TOO_SMALL
 cmp ecx, [.esi.Parameters.DeviceIoControl.OutputBufferLength]
 ja .error

 push ecx
 mov esi, [GDTR.Address]
 add esi, eax
 mov edx, edi   ; save EDI in EDX
 mov edi, [.edi.AssociatedIrp.SystemBuffer]
 rep movsb
 mov edi , edx   ; restore EDI

 pop [.edi.IoStatus.Information]

 mov [status], STATUS_SUCCESS
 jmp .exit

.next:

 jmp .exit
.error:
 mov [.edi.IoStatus.Information], 0
 jmp .exit

.exit:
 push [status]
 pop [.edi.IoStatus.Status]

 fastcall IofCompleteRequest, edi, IO_NO_INCREMENT
 mov eax, [status]

 pop edx
 pop ecx
 pop esi
```

```
    pop edi
    ret

endp
proc DispatchCreate pDeviceObject, pIrp

   mov eax, [pIrp]
   virtual at eax
    .eax IRP
   end virtual

   mov [.eax.IoStatus.Status], STATUS_SUCCESS
   and [.eax.IoStatus.Information], 0

   mov ecx, [pIrp]
   mov edx, IO_NO_INCREMENT
   call [IofCompleteRequest]

   mov eax, STATUS_SUCCESS
   ret

endp

proc DispatchClose pDeviceObject, pIrp

   mov eax, [pIrp]
   virtual at eax
    .eax IRP
   end virtual

   mov [.eax.IoStatus.Status], STATUS_SUCCESS
   and [.eax.IoStatus.Information], 0

   mov ecx, [pIrp]
   mov edx, IO_NO_INCREMENT
   call [IofCompleteRequest]

   mov eax, STATUS_SUCCESS
   ret

endp

proc DriverUnloadHandler pDriverObject

   invoke IoDeleteSymbolicLink, SymbolicLinkName
   invoke IoDeleteDevice, [DeviceObject]
   ret
endp
```

```
proc DriverEntry pDriverObject, pusRegistryPath

local status:DWORD

   mov [status], STATUS_DEVICE_CONFIGURATION_ERROR

   invoke IoCreateDevice, [pDriverObject], 0, DeviceName, FILE_DEVICE_
UNKNOWN, 0, FALSE, DeviceObject

   cmp eax, STATUS_SUCCESS
   jnz .END
   invoke IoCreateSymbolicLink, SymbolicLinkName, DeviceName
   cmp eax, STATUS_SUCCESS
   jnz .simlinkfail
   mov eax, [pDriverObject]

   virtual at eax
    .eax DRIVER_OBJECT
   end virtual

   mov [.eax.MajorFunction+IRP_MJ_CREATE*4], DispatchCreate
   mov [.eax.MajorFunction+IRP_MJ_CLOSE*4], DispatchClose
   mov [.eax.MajorFunction+IRP_MJ_DEVICE_CONTROL*4], DispatchControl
   mov [.eax.DriverUnload],  DriverUnloadHandler

   mov [status], STATUS_SUCCESS
   jmp .END
   .simlinkfail:
   invoke IoDeleteDevice, [DeviceObject]
   .END:

   mov eax, [status]
   ret

endp

section '.relocs' fixups readable writeable discardable

section '.idata' import readable writeable

  library ntoskrnl,'ntoskrnl.exe'

  import ntoskrnl,\
    IoCreateDevice, 'IoCreateDevice',\
    IoCreateSymbolicLink, 'IoCreateSymbolicLink',\
    IoDeleteDevice, 'IoDeleteDevice',\
    IoDeleteSymbolicLink, 'IoDeleteSymbolicLink',\
    IofCompleteRequest, 'IofCompleteRequest'
```

Начнём сначала – с функции DriverEntry. В начале её выполнения создаётся объект-устройство при помощи функции IoCreateDevice. Функция IoCreateDevice принимает семь параметров:

1. Указатель на объект-драйвер, созданный системой (PDRIVER_OBJECT).
2. Произвольный размер области дополнительной памяти устройства (device extension), которую можно выделить в каждом объекте устройство (DWORD).
3. Имя устройства (PUNICODE_STRING).
4. Уникальный идентификатор типа устройства (DWORD).
5. Дополнительная информация об устройстве (DWORD).
6. Возможность монопольного доступа к устройству (BOOL).
7. Указатель на переменную, в которую будет сохранён указатель на созданный объект-устройство.

Второй параметр имеет смысл использовать, если драйвер управляет несколькими устройствами. Таким образом, он может хранить относящуюся к каждому устройству информацию в самом объекте-устройстве. В нашем случае нет смысла в использовании дополнительной области памяти.

Уникальный идентификатор типа устройства необходим для того, чтобы система «знала» тип созданного устройства. Мы не будем ничего придумывать и поставим FILE_DEVICE_UNKNOWN, т. е. «неизвестный тип».

Шестой параметр нужен для указания дополнительных характеристик устройства; в нашем случае в этом нет необходимости.

После создания объекта-устройства необходимо создать символьную ссылку на него, чтобы код третьего кольца смог получить к нему доступ. Осуществляет данное действие функция IoCreateSymbolicLink. Функция принимает два параметра: имя символьной ссылки и имя устройства.

Следует заметить, что для объявления структур UNICODE_STRING был использован специальный макрос UNICODE_STRING_define – он максимально упрощает объявление строк в данном формате.

После создания символьной ссылки происходит заполнение массива MajorFunction (листинг 3.22).

Листинг 3.22. Заполнение массива MajorFunction

```
mov eax, [pDriverObject]

virtual at eax
 .eax DRIVER_OBJECT
end virtual

mov [.eax.MajorFunction+IRP_MJ_CREATE*4], DispatchCreate
mov [.eax.MajorFunction+IRP_MJ_CLOSE*4], DispatchClose
mov [.eax.MajorFunction+IRP_MJ_DEVICE_CONTROL*4],DispatchControl
mov [.eax.DriverUnload],  DriverUnloadHandler
```

Сначала указатель на объект-драйвер заносится в регистр EAX. После чего с помощью директивы virtual мы говорим компилятору, что регистр EAX указывает

на структуру DRIVER_OBJECT. Структура DRIVER_OBJECT содержит массив указателей MajorFunction, каждый элемент в котором содержит указатель на функцию-обработчик разных типов пакетов запроса ввода-вывода. Каждый элемент этого массива соответствует своему типу IRP. Если, например, драйверу необходимо обрабатывать запрос типа IRP_MJ_SHUTDOWN, уведомляющий о завершении работы системы, то он должен поместить в соответствующую позицию массива MajorFunction указатель на функцию, которой запросы этого типа и будут направляться. Если такая функциональность драйверу не нужна, как в нашем случае, то и заполнять этот элемент массива MajorFunction не требуется. Мы совсем не обязаны заполнять все элементы массива MajorFunction, коих в DDK определено целых 28 штук (IRP_MJ_MAXIMUM_FUNCTION + 1). Все зависит от задач, стоящих перед драйвером. В элементы массива MajorFunction, не заполненные драйвером, диспетчер ввода-вывода заносит указатель на внутреннюю функцию IopInvalidDeviceRequest. Эта функция уведомляет о попытке обращения к неподдерживаемой данным драйвером функции. В нашем случае происходит заполнение только тех элементов массива, которые обеспечивают успешный вызов функций CreateFile, DeviceIoControl и CloseHandle кодом режима пользователя.

В ntddk.inc среди прочих определены константы, представляющие для нас интерес, а именно:

```
IRP_MJ_CREATE                   equ 0
. . .
IRP_MJ_CLOSE                    equ 2
IRP_MJ_READ                     equ 3
IRP_MJ_WRITE                    equ 4
. . .
IRP_MJ_DEVICE_CONTROL           equ 0Eh
. . .
IRP_MJ_CLEANUP                  equ 12h
```

Это порядковые номера, определяющие положение указателя на процедуру диспетчеризации в массиве MajorFunction. Умножив соответствующую константу на размер указателя, равный 4 байтам, мы получим смещение в массиве MajorFunction, по которому должен быть расположен указатель на соответствующую процедуру-обработчик.

Теперь следует сказать несколько слов по поводу обработчиков запросов. Как и любые функции обратного вызова (т. е. те функции, которые вызываются самой системой), они должны отвечать соглашению передачи параметров stdcall. Все они принимают два параметра: первый параметр – указатель на объект-устройство, второй – указатель на структуру IRP, которая представляет запрос ввода-вывода.

Еще одно ключевое поле структуры DRIVER_OBJECT – это поле DriverUnload. Туда мы должны поместить (если хотим иметь возможность динамически выгружать драйвер) указатель на процедуру, которую система будет вызывать при обращении кода режима пользователя к функции ControlService с параметром SERVICE_CONTROL_STOP.

Если всё прошло успешно, необходимо возвратить системе код успеха STATUS_SUCCESS.

Следующий шаг – это обработчики запросов IRP_MJ_CREATE и IRP_MJ_CLOSE. Эти запросы генерируются при вызове кодом третьего кольца функций CreateFile и CloseHandle (или NtCreateFile и NtClose в нулевом кольце). В обработчике запроса IRP_MJ_CREATE драйвер должен подготавливать данные для дальнейшей работы с устройством; соответственно, в обработчике запроса IRP_MJ_CLOSE драйвер должен производить действия, противоположные действиям в обработчике IRP_MJ_CREATE. Обработчиками этих запросов являются функции DispatchCreate и DispatchClose; их код полностью идентичный (листинг 3.23).

Листинг 3.23. Функция обработки запроса IRP_MJ_CREATE

```
proc DispatchCreate pDeviceObject, pIrp

   mov eax, [pIrp]
   virtual at eax
    .eax IRP
   end virtual

   mov [.eax.IoStatus.Status], STATUS_SUCCESS
   and [.eax.IoStatus.Information], 0

   mov ecx, [pIrp]
   mov edx, IO_NO_INCREMENT
   call [IofCompleteRequest]

   mov eax, STATUS_SUCCESS
   ret

endp
```

Оба обработчика почти ничего не делают. Они всего лишь возвращают код успеха и уведомляют диспетчер ввода-вывода об успехе.

Вызов функции IofCompleteRequest инициирует операцию завершения ввода-вывода. Как следует из самого названия, эта функция уведомляет диспетчер ввода-вывода о том, что операция ввода-вывода завершена и можно отправлять ее результаты инициатору запроса (в нашем случае коду режима пользователя). О том, обработку какого именно запроса следует завершить, говорит первый параметр. Второй параметр определяет повышение уровня приоритета потока, инициировавшего операцию, после её завершения.

Если драйвер обслуживает физическое устройство, то операция ввода-вывода может длиться ощутимое время. При синхронных операциях ввода-вывода, как в нашем случае, пока поток ждет завершения операции, система не предоставляет ему процессорное время. По окончании операции ожидавший поток вправе немедленно возобновить выполнение и обработать полученные данные. Именно через второй параметр функции IofCompleteRequest драйвер и сообщает системе, на какую величину повысить приоритет ожидавшего потока. В нашем примере повышения приоритета потока не требуется, и мы передаем константу IO_NO_INCREMENT равную нулю.

Функция `IofCompleteRequest` является fastcall-функцией. В префиксе имени функции присутствует символ `f`. Существует, кстати сказать, и stdcall-вариант – `IoCompleteRequest` (обратите внимание на отсутствие символа `f` в префиксе). Но, в образовательных целях мы будем использовать быструю версию. Эта не единственная fastcall-функция – есть и другие. У них также есть свои stdcall-аналоги, которые, как правило, просто вызывают свои fastcall-аналоги.

Функции, отвечающие соглашению fastcall, принимают первый аргумент в регистре ECX, второй – в EDX; остальные аргументы, при наличии таковых, помещаются в стек в обратном порядке (справа налево, как stdcall). Стек очищает вызванная функция.

В коде функций `DispatchCreate` и `DispatchClose` вызов функции `IofCompleteRequest` осуществляется вручную – через занесение параметром в регистры, но для удобства лучше воспользоваться макросом, приведённым в листинге 3.24.

Листинг 3.24. Макрос для вызова fastcall-функций

```
macro fastcall func, p1, p2, [px]
{
   if ~ px eq
     reverse push px
   end if
   if ~ p1 eq
     mov ecx,p1
   end if
   if ~ p2 eq
     mov edx,p2
   end if
   call [func]
}
```

Вооружившись вышеприведённым макросом, вызывать fastcall-функции так же просто, как и stdcall-функции.

Итак, обработчики открытия и закрытия устройства написаны; остался один – обработчик запроса `IRP_MJ_DEVICE_CONTROL` (листинг 3.25).

Листинг 3.25. Функция-обработчик запроса IRP_MJ_DEVICE_CONTROL

```
proc DispatchControl pDeviceObject, pIrp
...
    cmp [.esi.Parameters.DeviceIoControl.IoControlCode], IOCTL_DUMP_GDT
    jz .IOCTL_DUMP
    jmp .next
   .IOCTL_DUMP:
    sgdt fword [GDTR]
    xor ecx, ecx
    mov cx, [GDTR.Limit]
    inc ecx ; ecx = GDT size

    xor eax, eax
```

```
    mov edx, [.esi.Parameters.DeviceIoControl.InputBufferLength]
    cmp edx, 8
    jnz .copydata
    mov eax, [.edi.AssociatedIrp.SystemBuffer]
    mov edx, [eax+4]  ; edx = needed data size
    mov eax, [eax]    ; eax = start offset in GDT

    add edx, eax
    cmp edx, ecx
    jna @f
    mov edx, ecx
   @@:
    sub edx, eax   ; edx = fact data size
    cmp edx, 7FFFFFFFh
    jna @f
    mov edx, 0
   @@:
    mov ecx, edx
   .copydata:
    mov [status], STATUS_BUFFER_TOO_SMALL
    cmp ecx, [.esi.Parameters.DeviceIoControl.OutputBufferLength]
    ja .error

    push ecx
    mov esi, [GDTR.Address]
    add esi, eax
    mov edx, edi   ; save EDI in EDX
    mov edi, [.edi.AssociatedIrp.SystemBuffer]
    rep movsb
    mov edi, edx   ; restore EDI

    pop [.edi.IoStatus.Information]

    mov [status], STATUS_SUCCESS
...
    push [status]
    pop [.edi.IoStatus.Status]
    fastcall IofCompleteRequest, edi, IO_NO_INCREMENT
    mov eax, [status]
...
    ret

endp
```

При получении управляющего кода `IOCTL_DUMP_GDT` происходит прыжок на метку `.IOCTL_DUMP`, за которым следуют обработка входных данных и передача необходимого участка GDT пользовательскому процессу.

Если размер входного буфера не равен 8, то пользовательскому процессу передаётся вся глобальная дескрипторная таблица. Если же во входном буфере

переданы данные, то они интерпретируются следующим образом: первое DWORD-значение задаёт смещение, начиная с которого будут копироваться данные, а второе DWORD-значение – размер копируемых данных. После извлечения входных данных производится их проверка, и в итоге при подходе к метке `.copydata` в обоих случаях в регистре EAX присутствует смещение, начиная с которого будут копироваться данные, а в регистре ECX указано количество байтов для копирования.

После выполнения всех операций и занесения в выходной буфер ответа на запрос необходимо занести в поле `IRP.IoStatus.Information` число байтов, реально скопированных в системный буфер, чтобы диспетчер ввода-вывода «знал», сколько байтов копировать в пользовательский выходной буфер. Также необходимо занести в поле `IRP.IoStatus.Status` результат обработчика запроса. Если результат равен `STATUS_SUCCESS`, то диспетчер ввода-вывода скопирует данные из системного буфера в пользовательский выходной буфер; в любом другом случае функция `DeviceIoControl` возвратит ошибку выполнения.

После обработки запроса необходимо уведомить диспетчер ввода-вывода о том, что операция ввода-вывода завершена и можно отправлять её результаты инициатору запроса. Для этого надо вызвать функцию `IofCompleteRequest`; в нашем примере вызов функции произведен при помощи макроса, который приводился выше при рассмотрении функций `DispatchCreate` и `DispatchClose`.

Последняя функция, которую мы ещё не рассмотрели, – `DriverUnloadHandler`. Её задача минимальна: освободить все данные и структуры, которые были созданы в функции `DriverEntry` либо во время работы драйвера. В нашем случае происходит удаление объекта-устройства и символьной ссылки на него.

В пакете FASM (который находится на компакт-диске, прилагающемся к книге) в папке INCLUDE находятся все заголовочные файлы, используемые программой управления драйвером и самим драйвером (папка \INCLUDE\DDK).

3.3.8.3. Отладка драйверов

Отладке драйверов режима ядра является более сложной задачей, нежели отладка приложений третьего кольца. В третьем кольце всё было просто: достаточно набросать примерный код, запустить под отладчиком, расставить точки останова или команды `int3` в критических местах – и дальше уже «в полевых условиях» разбираться, где и что неправильно работает. Если что-то пошло не так – мы завершаем процесс, перекомпилируем, снова запускаем, и т. д., до тех пор пока код не заработает как надо. При программировании драйверов об этой практике можно забыть: в нулевом кольце «сапер ошибается один раз». Одно неверное движение – и мы увидим «голубой экран смерти» (BSOD). Взявшись за разработку драйверов режима ядра, приготовьтесь к тому, что BSOD довольно часто будет появляться на экране вашего монитора!

Для отладки драйверов в режиме ядра потребуется отладчик режима ядра. Таких отладчиков немного: WinDBG, SoftICE и Syser Kernel Debugger. Отладчик WinDBG из пакета Debugging Tools for Windows, разработчиком которого является компания Microsoft, обладает широчайшим набором функций и возможностей, но вместе с тем имеет один существенный минус: отлаживать ядро с помощью

него можно, только имея в распоряжении два компьютера (отлаживаемый и тот, с которого производится отладка). Правда, не имея в распоряжении двух компьютеров, можно воспользоваться виртуальной машиной и отлаживать драйверы на ней. Тем не менее сам факт необходимости двух систем доставляет некоторые неудобства при отладке драйверов.

Отладчик SoftICE – самый распространенный. Он обладает широким функционалом и позволяет отлаживать систему без использования дополнительного компьютера. Но у SoftICE также есть серьёзный недостаток: этот отладчик уже более семи лет не поддерживается разработчиком, а потому не работает на новых версиях операционных систем Windows начиная с Windows XP SP3.

Отладчик Syser является платным и по функционалу мало чем уступает SoftICE. При этом он обновляется и поддерживается разработчиком (подходит и для новых версий операционных систем). Syser Kernel Debugger – наиболее разумная альтернатива SoftICE.

На компакт-диске, прилагающемся к данной книге, в папке debuggers есть trial-версия отладчика Syser Kernel Debugger и пакет Debugging Tools for Windows (WinDBG).

При отладке драйверов в нулевом кольце незаменимой является команда `INT3`, т. к. очень трудно узнать, по какому адресу происходит загрузка драйвера в память. При использовании команды `int3` достаточно вставить её в функцию `DriverEntry` (или в другом месте, которое необходимо трассировать). После этого при загрузке драйвера отладчик автоматически остановит его выполнение в точке входа.

3.3.9. Резюме

В данном разделе были рассмотрены основы программирования драйверов в Win32. Однако изложенный здесь материал является лишь верхушкой айсберга – за пределами рассмотрения осталось очень много аспектов программирования драйверов, такие как асинхронный ввод-вывод, метод управления буфером при прямом вводе-выводе, взаимодействие с файловой системой, реестром, системной памятью и др. На полное изложение всех аспектов программирования в ядре операционных систем Windows не хватило бы всей этой книги.

4 LONG MODE

4.1. Введение в long mode

В предыдущих разделах мы говорили о 32-битном программировании: о программировании в защищённом режиме и программировании под операционной системой защищённого режима Windows. В этом и следующем разделе мы будем говорить о 64-битном программировании: программировании в 64-разрядном режиме процессора (long mode) и программировании в 64-битных версиях операционных систем Windows.

Технология AMD64 – это 64-битная архитектура микропроцессора и соответствующий набор инструкций, разработанный компанией AMD. AMD64 – это расширение архитектуры x86 под 64-рязрядную архитектуру, с полной обратной совместимостью. Эти архитектура и набор инструкций были лицензированы (с незначительными изменениями) основным конкурентом AMD – компанией Intel – под названием EM64T. Такой шаг компании Intel был более чем правильным, т. к. две различные 64-разрядные архитектуры привели бы только к замедлению прогресса в этой области, в частности при разработке 64-разрядных операционных систем.

Согласно документации от компании Intel 64-битный режим процессора официально называется IA-32e, а согласно документации от компании AMD – long mode. Далее термином «long mode» будем обозначать сам 64-битный режим как таковой, с обоими его подрежимами. Под словами «64-битный режим» будем понимать именно тот подрежим, в котором выполняется 64-разрядный код.

4.1.1. Общий обзор

Режим long mode имеет два подрежима: собственно сам 64-битный режим и режим совместимости. В 64-битном режиме размер адреса по умолчанию равен 8 байтам, размер операнда по умолчанию равен 4 байтам; в режиме совместимости размер адреса равен 4 байтам, размер операнда по умолчанию – 4 байтам, так же как и в защищённом режиме. Режим совместимости включается путём установки специального бита в дескрипторе сегмента кода.

В 64-битном режиме процессора доступны следующие регистры процессора:

- регистры общего назначения: 64-разрядные RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP и R8,R9,...,R15; 32-разрядные EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP; R8D–R15D являются младшими половинами 64-разрядных регистров; 16-разрядные AX, BX, CX, DX, SI, DI, SP, BP, R8W–R15W являются младшими частями 32-разрядных регистров; 8-битные регистры AH, BH, CH, DH и AL, BL, CL, DL, SIL, DIL, SPL, BPL, R8L–R15L – старшие и младшие части 16-битных регистров соответственно;

- 64-разрядный RIP – указатель инструкции;
- 16-разрядные сегментные регистры: CS, DS, SS, ES, FS, GS;
- 64-разрядный регистр флагов – RFLAGS;
- 80-битные регистры математического сопроцессора ST0–ST7;
- 64-битные MMX-регистры (MM0–MM7);
- 128-разрядные XMM-регистры – XMM0–XMM15 и 32-битный MXCSR.;
- 64-разрядные регистры управления CR0–CR4 и CR8; регистры-указатели системных таблиц GDTR, LDTR, IDTR и регистр задачи TR;
- 64-разрядные регистры отладки – DR0–DR3, DR6, DR7;
- MSR-регистры.

Система команд в long mode почти не претерпела сильных изменений, формат команд остался тем же. Доступ к 64-битным регистрам и новым регистрам осуществляется через специальный REX-префикс. Таким образом, все опкоды команд, которые работают с 64-битными регистрами, увеличиваются в размере как минимум на 1 байт, и возникает серьёзная проблема оптимизации кода. Поэтому рекомендуется везде, где возможно, использовать 32-битные регистры.

Команды PUSHAD/POPAD по понятным причинам теперь уже не поддерживаются. Команды PUSH и POP могут работать только с 64- и 16-битными операндами; при помещении в стек 16-битный операнд не расширяется до 64 бит, но при этом указатель на верхушку стека перестаёт быть выровненным на 8 байт.

Регистр EFLAGS расширен до 64 бит и теперь называется RFLAGS, но никаких новых полей в нём не появилось. Также из-за введения 64-битной системы команд теперь стало проще работать с 64-битными числами, и некоторые операции можно произвести одной командой.

В системе команд 64-битного режима усложнилось использование команд дальней передачи управления. Теперь команды типа `JMP selector16:offset64` просто не существуют. Для межсегментной передачи управления (обычно она используется для перезагрузки регистра CS) надо использовать способ, приведённый ниже в листинге 4.1.

Листинг 4.1. Дальний прыжок в long mode

```
jump_value:
dq jump_offset
dw jump_selektor
...
jmp tbyte[jump_value]
```

Такой способ был доступен и в защищённом режиме, но в 64-битном режиме он является единственно возможным.

Кроме того, нельзя использовать команду следующего формата: `mov [address], value64`. 64-битное значение можно заносить только в регистр (в то время как в защищённом режиме можно было использовать `mov [address], value32`). Если необходимо занести 64-битное значение в ячейку памяти, надо начала занести его в регистр, а потом из регистра – в память; таким образом, ряд действий, которые в защищённом режиме выполнялись за один шаг, теперь будет выполняться в два шага.

В 64-битном режиме введён режим адресации относительно RIP. В случае адресации относительно RIP используются не 64-битные адреса данных и переходов, а 32-битные. Таким образом, можно достигнуть почти 40%-ного уменьшения размера кода. Использование адресации относительно RIP почти не зависит от программиста, а зависит только от используемых им команд. В настоящей редакции книги такой способ оптимизации программ рассматриваться не будет.

Значительные изменения в long mode претерпела сегментация. Сегментация в long mode поддерживается только частично: все сегменты начинаются с адреса 0h и покрывают всё 64-битное адресное пространство, проверка лимита не производится. В дескрипторах данных игнорируются почти все поля, в дескрипторе кода игнорируется большинство полей. Также в long mode механизм многозадачности не поддерживается аппаратно – его надо реализовывать программно.

Небольшие изменения претерпел механизм трансляции страниц, в большей степени из-за того, что увеличилось количество виртуальной памяти. Теперь фактически доступно до 2^{48} байт (256 Тб) виртуальной памяти и 2^{52} байт физической.

4.1.2. Сегментация в long mode

Как уже было сказано, сегментация в long mode фактически отключена: проверка лимита сегментов не осуществляется, все сегменты кода и данных имеют базовый адрес равный нулю. В целях поддержки совместимости в long mode можно получить доступ к теневым частям сегментных регистров FS и GS через MSR-регистры `IA32_FS_BASE` и `IA32_GS_BASE`. Таким образом, имеется возможность изменить базовый адрес для сегментов, описываемых в FS и GS, что ликвидирует возможные проблемы при переносе 32-битных систем на 64-битную платформу (например, в системах Win32 регистр FS указывает на блок информации о текущем потоке).

На рис. 4.1 приведена структура дескриптора сегмента кода в long mode; серым цветом помечены поля, которые игнорируются процессором:

Как видно из рисунка, поля лимита и базового адреса игнорируются – остались только наиболее важные поля. назначение всех полей такое же, как и в защищённом режиме. Следует обратить внимание на поле L и поле D. Если поле L равно нулю, процессор находится в режиме совместимости – вернее, код, который выполняется в сегменте, описываемом данным дескриптором, выполняется в режиме совместимости. Если бит D сброшен, то размер операнда по умолчанию равен 16 битам (16-битный код); если он выставлен, то размер операнда по умолчанию равен 32 битам. Если бит L равен единице, то процессор работает в 64-битном

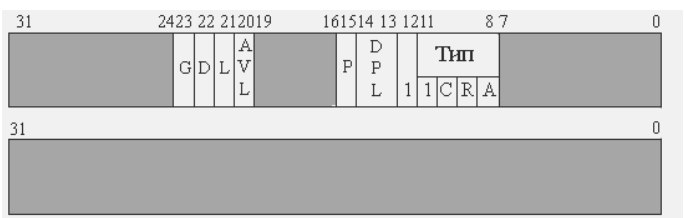


Рис. 4.1. Дескриптор сегмента кода в long mode

режиме. Комбинация, когда биты D и L равны единице одновременно, зарезервирована для дальнейшего использования.

Аналогичная ситуация наблюдается с дескрипторами сегментов данных. В 64-битном режиме из-за того, что не производится проверка лимитов сегментов, игнорируются поля, связанные с начальным адресом и лимитом, а именно: поля базового адреса, лимита, бит G.

Следует отметить, что на процессорах AMD игнорируется ещё больше битов, без которых в принципе можно обойтись. На процессорах AMD в дескрипторах сегментов данных игнорируются все поля кроме `Present`, а в дескрипторах сегментов кода – поля `R`, `A`, `G` (непонятно, почему они не игнорируются на процессорах Intel). Нельзя об этом забывать при проектировании 64-битных операционных систем и программ.

4.1.3. Механизм трансляции страниц

В long mode есть только один режим трансляции страниц – режим PAE. Он всегда должен быть включён; перейти в long mode при отключённом режиме PAE нельзя, и отключить его тоже нельзя. Как уже было сказано, в 64-битном режиме доступны 2^{48} байт виртуальной памяти и до 2^{52} байт физической памяти.

Страницы могут быть размером 4 Кб, 2 Мб и 1 Гб, причём допускается их комбинирование. При трансляции адреса используются 4 типа системных таблиц. Помимо таблиц страниц, каталогов страниц, таблиц указателей на каталоги страниц добавлена ещё одна таблица – карта страниц четвёртого уровня (PML4). Регистр CR3 расширяется до 8 байт для возможности хранения 64-битного адреса PML4. Более подробно о механизме трансляции страниц будет рассказано в следующих главах.

4.1.4. Переход в long mode

Переключение процессора в 64-битный режим сложнее, чем переход из режима реальных адресов в защищённый режим. Теперь помимо подготовки глобальной дескрипторной таблицы необходимо подготовить таблицы для трансляции адресов.

Переходом в long mode управляет регистр MSR под названием EFER (в документации Intel он фигурирует как IA32_EFER). Этот регистр находится по адресу 0C0000080h. В нём есть два важных поля – это биты под номерами 8 и 10. Бит под номером 8 носит название LME – именно ему надо присвоить единицу, для того чтобы включить long mode. Бит под номером 10 называется LMA; он говорит процессору, чтобы он работал в 64-битном режиме. Если бит LME выставлен, это не означает, что процессор будет работать в 64-битном режиме: с этой целью надо выставить бит LMA. Бит LME доступен для чтения и записи, бит LMA – только для чтения.

Итак, для того чтобы правильно перейти в 64-битный режим процессора, необходимо проделать следующие действия:

1. Выключить механизм трансляции страниц (бит 31 в регистре CR0) и включить режим PAE (бит 5 в регистре CR4).

2. Загрузить в регистр CR3 указатель на карту страниц четвёртого уровня (PML4).
3. Выставить бит LME (бит LMA доступен только для чтения) в MSR-регистре EFER.
4. Включить механизм трансляции страниц. Выставить бит PG в регистре CR0. Именно это действие побудит процессор присвоить полю LMA в регистре EFER значение 1, т. е. включить бит LMA.
5. Следующей командой надо выполнить команду JMP, в качестве селектора указав селектор на дескриптор кода, в котором выставлен бит L.

После того как будет включён механизм трансляции страниц, процессор будет находиться в режиме совместимости, т. к. регистр CS указывает на дескриптор 32-битного сегмента кода. В принципе мы уже находимся в long mode, но для окончательного завершения операции остаётся только загрузить в регистр CS селектор, указывающий на 64-битный дескриптор сегмента кода.

4.1.5. Практика

Основные принципы 64-битного режима процессора мы изучили; все необходимые знания для написания программы, которая переводит процессор в 64-битный режим, тоже получили – осталось только написать эту программу.

Итак, сейчас мы напишем программу, которая переводит процессор в long mode и выводит сообщение на экран. Код перевода процессора в защищённый режим остался тем же, поэтому приводить его здесь не имеет смысла. Поскольку у нас будет ещё как минимум три программы в long mode, то и эту первую программу 64-битного режима мы построим по тому же принципу, что и наши программы, работающие в защищённом. Мы напишем один раз код, который переводит процессор в long mode и в конце просто включим файл с кодом, который выводит сообщение (листинг 4.2).

Листинг 4.2. Перевод процессора в 64-битный режим

```
.................
align 8
GDT:
     NULL_descr    db      8 dup(0)
     CODE32_descr  db      0FFh,0FFh,00h,00h,00h,10011010b,11001111b,00h
     DATA_descr    db      0FFh,0FFh,00h,00h,00h,10010010b,11001111b,00h
     CODE64_descr  db      00h, 00h,00h,00h, 00h,10011000b,00100000b,00h
     GDT_size      equ     $-GDT

label GDTR fword
          dw       GDT_size-1
          dd       ?

use32
PROTECTED_MODE_ENTRY_POINT:
     mov ax, DATA_SELEKTOR
```

```
      mov ds, ax
      mov es, ax
      mov ss, ax
      mov esp, STACK_BASE_ADDRESS

      call delta
   delta:
      pop ebx
      add ebx, PM_CODE_START-delta

      mov esi, ebx
      mov edi, PM_CODE_BASE_ADDRESS
      mov ecx, PM_CODE_SIZE
      rep movsb

      mov eax,PM_CODE_BASE_ADDRESS
      jmp eax

PM_CODE_START:
ORG PM_CODE_BASE_ADDRESS

      mov eax, cr4
      bts eax, 5   ; PAE = 1
      mov cr4, eax

      mov dword [PDE_addr], 010000011b;PS or Present or Write
      mov dword [PDE_addr+4], 0
      mov dword [PDPE_addr], PDE_addr or 3 ; Present or Write
      mov dword [PDPE_addr+4], 0
      mov dword [PML4_addr], PDPE_addr or 3; Present or Write
      mov dword [PML4_addr+4], 0

      mov eax, PML4_addr
      mov cr3, eax

      mov ecx, 0xC0000080 ; EFER
      rdmsr
      bts eax,8  ; EFER.LME = 1
      wrmsr

      mov eax, cr0
      bts eax, 31 ; PG = 1
      mov cr0, eax

      jmp CODE64_SELEKTOR:LONG_MODE_ENTRY_POINT
use64
LONG_MODE_ENTRY_POINT:
      mov ax, ds  ; перезагрузить сегментные регистры
      mov ds, ax
```

```
        mov ss, ax
        mov es, ax

include 'LM_CODE.ASM'
PM_CODE_END:
```

После перехода в защищённый режим происходит перемещение основного его кода по адресу PM_CODE_BASE_ADDRESS. После прыжка на адрес PM_CODE_BASE_ADDRESS сначала выставляется бит PAE в регистре CR4. После установки бита PAE мы подготавливаем структуры для механизма трансляции страниц.

Сам механизм трансляции адресов в этом разделе рассматриваться не будет (к этой теме мы обратимся в разделе 4.2), поэтому код, который подготавливает структуры для механизма трансляции адресов, надо принять как «набор инструкций». Корневой таблицей, использующейся при трансляции адресов в long mode, является карта страниц четвёртого уровня (PML4). После загрузки адреса PML4 в регистр CR3 мы устанавливаем бит LME в регистре IA32_EFER. Потом включаем страничную адресацию (выставляем бит PG в регистре CR0) и производим прыжок на 64-битный код. Перезагрузку сегментных регистров не стоит принимать как обязательное действие – это делается лишь «на всякий пожарный случай». И под конец мы включаем файл с кодом, который выводит сообщение о том, что мы перешли в long mode. Его содержимое представлено в листинге 4.3.

Листинг 4.3. Код, выполняющийся в 64-битном режиме

```
LM_CODE_START:
   mov rsi, message2
   mov rdi, 0B8000h
   mov rcx, mess2end-message2
   rep movsb
   jmp $

message2 db "W5e5 5i5n5 5l5o5n5g5 5m5o5d5e5!5"
mess2end:
```

Полный исходный код файлов LM.ASM и LM_CODE.ASM находится на диске в папке part4. Так же как и программы, описанные в главе 2, программу, переключающую процессор в 64-битный режим, следует запускать из под MS-DOS, при этом произведя загрузку либо с загрузочной «флешки», либо с дискеты.

4.1.6. Резюме

В этом разделе мы познакомились с 64-битным режимом процессора, изучили основные аспекты работы в этом режиме, также написали программу, переводящую процессор в 64-разрядный режим (или long mode). В следующем разделе будет описана работа с памятью в 64-битном режиме.

4.2. Работа с памятью в long mode

В разделе 4.1 мы говорили об основах 64-разрядного режима процессора. В этом разделе будут излагаться принципы работы с виртуальной памятью в 64-битном

режиме процессора. Работа с памятью в long mode – это второй по важности аспект работы в этом режиме, т. к. 64-битный режим может работать только с включённым механизмом трансляции страниц и почти вся защита данных и кода операционной системы теперь почти полностью возлагается на уровень страниц.

Работа с виртуальной памятью стала немного сложнее, чем в защищённом режиме, поскольку увеличилось количество таблиц и структур для описания страниц памяти, но основные принципы работы с виртуальной памятью не изменились.

4.2.1. Общий обзор

В 64-битном режиме доступно до 2^{52} байт физической памяти и 2^{48} байт виртуальной памяти. Максимальная разрядность физического адреса равная 52 поддерживается не всеми моделями процессора (скорее всего, она будет меньше); для получения максимальной разрядности физического адреса, поддерживаемой процессором, на котором выполняется программа, следует использовать инструкцию CPUID.

Следует отметить, что ограничение на разрядность физического адреса равную 52 является ограничением самой архитектуры процессора. Не следует ждать, что новые процессоры будут поддерживать более 2^{52} байт памяти (что впрочем, и так много). Что же касается виртуального адреса, ограничение в 48 бит не является таким жёстким ограничением, как ограничение на разрядность физического адреса; возможно, что в будущих моделях процессоров будут введены дополнительные структуры для трансляции виртуальных адресов более 48 бит.

В 64-битном режиме доступны следующие размеры страниц: 4 Кб (виртуальный адрес делится на пять частей) и 2 Мб (виртуальный адрес делится на четыре части). Также есть возможность использовать страницы размером 1 Гб (виртуальный адрес делится на три части), но они поддерживаются не всеми процессорами. Кроме того, в long mode флаг PSE в регистре CR4 игнорируется и уже ни на что не влияет.

Как уже было сказано, в long mode может быть использовано 2^{48} байт виртуальной памяти. Все используемые виртуальные адреса должны быть в канонической форме. Это означает, что все старшие 16 бит адреса могут быть либо нулями, либо единицами. В любом другом случае мы получим исключение общей защиты (#GP) либо ошибку стека (#SS). Де-факто ограничение канонической формы адреса ведёт к тому, чтобы старшие 16 бит виртуального адреса не использовались вообще.

Каноническая форма адреса является мерой для обеспечения совместимости с будущими моделями процессоров. Связано это с тем, что большинство операционных систем делят адресное пространство на две части: в младшей части – память обычных процессов, в старшей – память ядра. Таким образом, запрет использовать старшие 16 бит виртуального адреса обеспечивает совместимость с будущими моделями процессоров, в которых эффективная часть виртуального адреса будет большей размерности.

4.2.2. Страницы размером 4 Кб

Размер страницы равный 4 Кб является самым ходовым; такой тип страниц обеспечивает сравнительно малую гранулярность распределения памяти. Также не следует забывать, что при использовании страниц такого размера процессор тратит много времени на трансляцию адреса.

При использовании страниц размером 4 Кб виртуальный адрес делится на пять частей:

1. Биты 47–39 – это индекс в карте страниц четвёртого уровня (PML4).
2. Биты 38–30 – это индекс в таблице указателей на каталоги страниц (PDPT).
3. Биты 29–21 – это индекс в каталоге страниц (PD).
4. Биты 20–12 – это индекс в таблице страниц (PT).
5. Биты 11–0 – это смещение на странице.

Каждая таблица, используемая при трансляции адреса, содержит 512 элементов размером 8 байт. Схематично механизм трансляции виртуального адреса в физический показан на рис. 4.2.

Далее на рис. 4.3–4.6 приведены форматы элементов каждой таблицы.

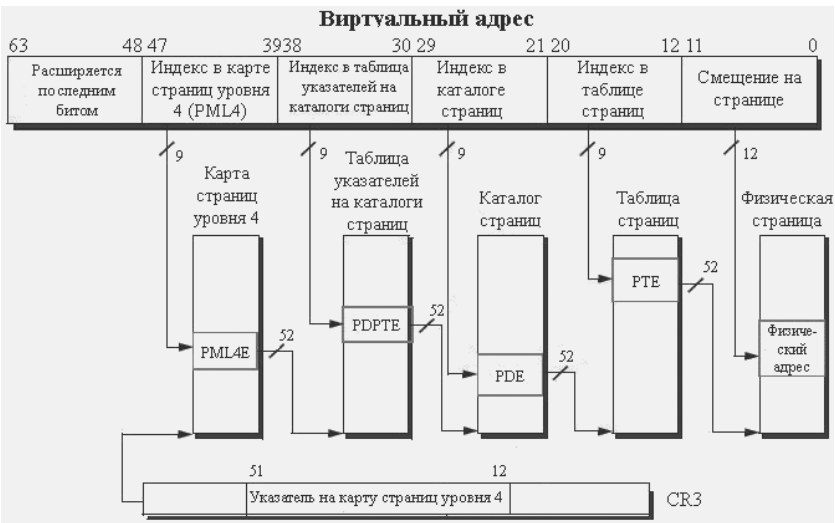


Рис 4.2. Трансляция адреса для страниц размером 4 Кб в long mode

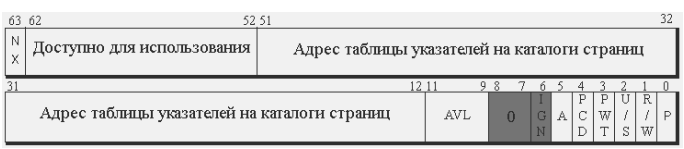


Рис. 4.3. Формат элемента в PML4 (PML4E)

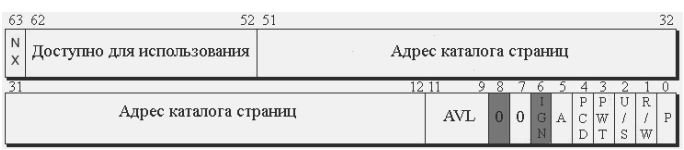


Рис. 4.4. Формат элемента таблицы указателей на каталоги страниц (PDPTE)

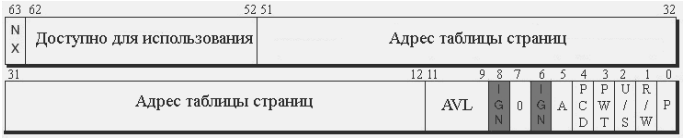


Рис. 4.5. Формат элемента каталога страниц (PDE) при использовании страницы размером 4 Кб

Рис. 4.6. Формат элемента таблицы страниц (PTE)

Назначение битов сходно с назначением одноимённых битов в элементах таблиц защищённого режима (см. раздел 2.3). Бит 7 в элементах таблиц (за исключением элемента PML4) есть не что иное, как бит PS. Этот бит сброшен в элементе таблицы указателей на каталоги страниц и в элементе каталога страниц. Именно это и говорит процессору, что используется страница размером 4 Кб.

4.2.3. Страницы размером 2 Мб

При использовании страниц размером 2 Мб виртуальный адрес делится уже не на пять частей, а на четыре части, в результате чего увеличивается скорость трансляции адреса. Использование страниц размером 2 Мб также на порядок уменьшает объём памяти, отводимый для таблиц, описывающих виртуальную память. Несмотря не указанные плюсы, у страниц размером 2 Мб есть один немаловажный минус – большая гранулярность выделения памяти.

Итак, перечислим четыре части виртуального адреса:

1. Биты 47–39 – это индекс в карте страниц четвёртого уровня (PML4).
2. Биты 38–30 – это индекс в таблице указателей на каталоги страниц.
3. Биты 29–21 – это индекс в каталоге страниц.
4. Биты 20–0 – это смещение на странице.

Таблицы страниц при использовании страниц такого размера не применяются; именно за счёт этого на порядок уменьшается размер памяти, отводимый под структуры, описывающие виртуальную память. На рис. 4.7 приведена схема трансляции адреса страницы размером 2 Мб.

Формат элемента карты страниц четвёртого уровня и таблицы указателей на каталоги страниц остался таким же – изменился формат элемента каталога страниц. Далее на рис. 4.8 приведён формат элемента каталога страниц.

Как видно из рисунка, поле PS (бит 7) равно единице; именно это и говорит процессору, что адрес, указанный в этом элементе, является адресом страницы, а не таблицы страниц.

Так же, как и в случае со страницами размером 4 Кб, пользовательский код будет иметь доступ к странице, только когда бит U/S выставлен в каждой структуре,

Рис. 4.7. Трансляция адреса для страниц размером 2 Мб в long mode

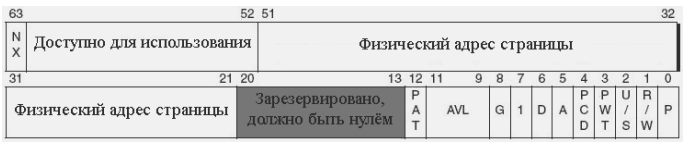


Рис. 4.8. Формат элемента каталога страниц (PDE) при использовании страницы размером 2 Мб

описывающей искомую страницу. Пользовательский код будет иметь возможность изменять данные только в том случае, когда бит R/W выставлен в каждой структуре, описывающей искомую страницу.

4.2.4. Страницы размером 1 Гб

Также в long mode поддерживаются страницы размером 1 Гб – хотя и не всеми процессорами. Узнать, поддерживает ли процессор страницы размером 1 Гб, можно, вызвав инструкцию CPUID и передав ей число 80000001h в регистре EAX. Двадцать шестой бит в регистре EDX показывает, поддерживает ли процессор страницы размером 1 Гб.

При использовании страниц размером 1 Гб виртуальный адрес делится на три части: индекс в карте страниц четвёртого уровня (биты 47–39), индекс в таблице указателей на каталоги страниц (биты 38–30) и смещение внутри страницы (биты 29–0). На рис. 4.9 приведена схема трансляции адреса страницы размером 1 Гб.

Формат элемента карты страниц четвёртого уровня не изменился – изменился только формат элемента таблицы указателей на каталоги страниц. На рис. 4.10 приведён формат элемента таблицы указателей на каталоги страниц.

Использование страниц размером 1 Гб является хорошим подспорьем при разработке операционных систем. У таких страниц есть очень хорошее преимущество:

Рис. 4.9. Трансляция адреса для страниц размером 1 Гб в long mode

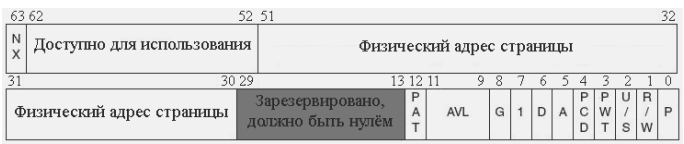


Рис. 4.10. Формат элемента таблицы указателей на каталоги страниц при использовании страниц размером 1 Гб (PDPTE)

используется всего две таблицы при трансляции адреса и, следовательно, ускоряется время самой трансляции адреса. Например, на странице размером 1 Гб можно размещать код ядра операционной системы, наиболее критические в плане скорости выполнения функции и наиболее часто использующиеся данные операционной системы.

4.2.5. Регистр CR3

В long mode регистр CR3 расширяется до 64 бит. Формат регистра CR3 изображён на рис. 4.11.

Биты 51–12 содержат физический адрес карты страниц четвёртого уровня (PML4). Поля PCD и PWN отвечают за кэширование страницы, на которой находится карта страниц четвёртого уровня.

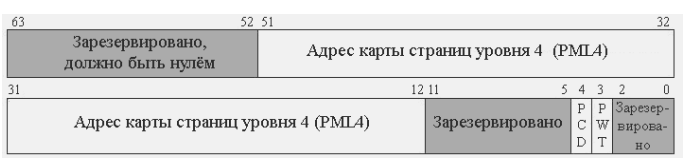


Рис. 4.11. Регистр CR3 в long mode

4.2.6. Проверки защиты

Правильное построение защиты на уровне страниц является залогом успеха 64-разрядной операционной системы; связано это в первую очередь с тем, что основные защитные механизмы в long mode теперь возлагаются на механизм страничного преобразования.

Защиту на уровне страниц в long mode осуществляют следующие биты:

1. Бит R/W в элементе каждой таблицы.
2. Бит U/S в элементе каждой таблицы.
3. Бит NX в элементе каждой таблицы.
4. Бит WP в регистре CR0.
5. Бит NXE в MSR регистре IA32_EFER.

Пользовательский код будет иметь доступ к странице только в том случае, когда бит U/S выставлен в каждой структуре, описывающей искомую страницу. Привилегированный код будет иметь доступ к странице независимо от того, выставлен ли бит U/S.

Пользовательский код будет иметь возможность изменять данные только в том случае, когда бит R/W выставлен в каждой структуре, описывающей данную страницу.

Так же, как и в защищённом режиме, привилегированный код по умолчанию (разумеется, если мы не изменяли шестнадцатый бит в регистре CR0) может изменять даже те страницы, которые помечены «только для чтения». Когда бит WP сброшен, привилегированный код (т. е. код, находящийся на уровнях 0, 1 и 2) имеет доступ на изменение ко всем страницам в системе. Если бит WP (шестнадцатый бит) в регистре CR0 выставлен, то привилегированный код работает по тем же правилам, что и пользовательский код, т. е. может изменить данные на странице только в том случае, когда бит R/W выставлен в каждой структуре, описывающей искомую страницу.

Код (пользовательский или привилегированный) сможет выполняться на странице, только когда бит NX сброшен в каждой структуре, описывающей искомую страницу. Действие флага NX не зависит от флага WP и одинаково влияет на пользовательский и привилегированный код. Проверка поддержки бита NX и включение этой возможности описывались в разделе 2.3.

4.2.7. Практика

Итак, мы изучили структуры и таблицы, используемые при работе с виртуальной памятью. Теперь следует немного попрактиковаться и написать программу, демонстрирующую возможности виртуальной памяти. Ниже будет представлена программа, которая отображает несколько виртуальных страниц на одну физическую и записывает туда данные. Для того чтобы иметь возможность понаблюдать работу нашей программы, мы будем проецировать виртуальные страницы на физическую страницу с адресом 0B8000h.

Все таблицы будут начинаться с определённого адреса; на первой странице, расположенной по этому адресу, будет таблица PML4, далее по мере надобности

будут создаваться нужные таблицы. Алгоритм выделения таблиц для описания виртуальной страницы следующий. В некоторой переменной будет храниться следующая свободная страница (отсчёт ведётся с PML4, и PML4 в начале обнуляется). При заполнении элемента в PML4, если у него сброшен бит P, обнуляем очередную страницу, порядковый номер которой указан в нашей переменной, и увеличиваем значение переменной на единицу. Если бит P выставлен, то просто переходим на страницу, которая указана в этом элементе. Аналогичные действия производятся и с остальными таблицами.

Приводить полный код примера не имеет смысла, поэтому далее приведён только код процедуры создания виртуальной страницы.

Листинг 4.4а. Функция создания виртуальной страницы

```
NextAvail4KPage dd 1  ; переменная, хранящая порядковый номер
  ; следующей свободной страницы
AllocPage_4K:
;IN
;   RAX физический адрес
;   RBX виртуальный адрес
   push rax
   push rbx
   push r8
   push rdi
   push rdx
   push r9
   push r10
   push r11
   push r12    ; сохраняем все используемые регистры

   mov r12, rax    ; r12 = физический адрес

   mov r8, rbx
   shr r8, 39  ; r8 = индекс в PML4

   shl r8, 3   ; r8 = смещение в PML4
   add r8, PLM4_BASE_ADDRESS ; r8 = PML4 entry addr
   mov r9d, [r8+4]
   shl r9, 32
   mov r9d, [r8]      ; r9 = PML4 entry
   mov eax, r9d       ; eax = r9d
   and eax, 1      ; проверка флага присутствия
   jz @f
   jmp .setPDPE
    @@:
```

Вначале мы сохраняем все регистры, которые будем использовать, в стеке. Потом сохраняем физический адрес в регистре R12, т. к. регистр RAX будет активно нами использоваться. После чего вычисляем адрес элемента в PML4 и проверяем

его флаг P (в регистре R8 хранится указатель на элемент в PML4). Если бит P не выставлен, то переходим к выделению таблицы указателей на каталоги страниц.

Листинг 4.4б. Функция создания виртуальной страницы *(продолжение)*

```
xor rdi, rdi
mov edi, [NextAvail4KPage]
shl rdi, 12   ; rdi = адрес следующей свободной страницы
add rdi, PLM4_BASE_ADDRESS
call ZeroPage_4K

mov rax, rdi  ; rax = адрес PDP-таблицы
or eax, 3     ; rax = addr or P or R/W
mov rdx, rax  ; rdx = rax

mov [r8], eax
shr rax, 32
mov [r8+4], eax
inc dword [NextAvail4KPage]
mov r9, rdx
```

Сначала происходит обнуление страницы с порядковым номером, указанным в переменной NextAvail4KPage. После обнуления страницы выставляются два младших бита в адресе страницы (биты P и R/W), и полученное значение заносится в элемент, указатель на который хранится в регистре R8. После чего индекс следующей свободной страницы увеличивается на единицу. После перехода на метку `.setPDE` в обоих случаях в регистре R9 находится содержимое элемента PML4.

Листинг 4.4в. Функция создания виртуальной страницы *(продолжение)*

```
.setPDPE:             ; r9 = содержимое элемента в PML4
;-------------------------------------------------------
and r9d, 0FFFFF000h  ; r9 =  адрес таблицы PDP
mov r8, rbx
shr r8, 30
and r8d, 1FFh      ; r8 = индекс  PDP
shl r8, 3          ; r8 = смещение в PDP
add r8, r9         ; r8 =  адрес элемента в PDP таблице
mov r10d, [r8+4]
shl r10, 32
mov r10d, [r8]     ; r10 = PDP entry
mov eax, r10d      ; eax = r10d
and eax, 1         ; проверка флага P
jz @f
jmp .setPDE
@@:
```

Во фрагменте кода, приведённом в листинге 4.4в, производятся практически те же действия, что и с элементом PML4. В R9 вначале находится содержимое элемента PML4, из него извлекается адрес, после чего извлекается содержимое элемента таблицы указателей на каталоги страниц и проверяется его бит P.

Листинг 4.4г. Функция создания виртуальной страницы *(продолжение)*

```
xor rdi, rdi
mov edi, [NextAvail4KPage]
shl rdi, 12   ; rdi = адрес следующей доступной страницы
add rdi, PLM4_BASE_ADDRESS
call ZeroPage_4K

mov rax, rdi
or eax, 3     ; rax = addr or P or R/W
mov rdx, rax

mov [r8], eax
shr rax, 32
mov [r8+4], eax
inc dword [NextAvail4KPage]
mov r10, rdx  ; r10 = PDP entry
```

В вышеприведённом листинге выделяется страница для каталога страниц, выставляются флаги и сохраняется значение в элемент таблицы указателей на таблицы страниц. После перехода на метку `.setPDE` в обоих случаях в регистре R10 будет находится содержимое элемента таблицы указателей на каталоги страниц.

Листинг 4.4д. Функция создания виртуальной страницы *(продолжение)*

```
.setPDE:              ; r10 = содержимое элемента таблицы PDP
;-------------------------------------------------------
and r10d, 0FFFFF000h ; r10 = адрес каталога страниц

mov rdx, rbx
shr rdx, 21
and edx, 1FFh;
xor r8, r8
mov r8d, edx          ; r8 = индекс в каталоге старниц
shl r8, 3             ; r8 = смещение в каталоге страниц
add r8, r10           ; r8 = адрес элемента в каталоге страниц
mov r11d, [r8+4]
shl r11, 32
mov r11d, [r8]
mov eax, r11d         ; eax = r10d
and eax, 1            ; проверка флага P
jz @f
jmp .setPTE
@@:
xor rdi, rdi
mov edi, [NextAvail4KPage]
shl rdi, 12    ; rdi = адрес следующей доступной страницы
add rdi, PLM4_BASE_ADDRESS
call ZeroPage_4K

mov rax, rdi      ; rax = адрес таблицы страниц
```

```
or eax, 3          ; rax = addr or P or R/W
mov rdx, rax

mov [r8], eax
shr rax, 32
mov [r8+4], eax
inc dword [NextAvail4KPage]
mov r11, rdx       ; r11 = PD entry
```

В листинге 4.4д производятся практически те же действия, что и с предыдущей таблицей. После перехода на метку в обоих случаях в регистре R11 будет находиться содержимое элемента в каталоге страниц.

Листинг 4.4е. Функция создания виртуальной страницы *(продолжение)*

```
.setPTE:
;-------------------------------------------------------
and r11d, 0FFFFF000h    ; r11 = PT addr

mov rdx, rbx
shr rdx, 12
and edx, 1FFh
xor r8, r8
mov r8d, edx       ; r8 = индекс в таблице страниц
shl r8, 3          ; r8 = смещение в таблице страниц
add r8, r11        ; r8 = адрес элемента таблицы страниц

                   ; в R12 находится физический адрес
or r12d, 3         ; addr or P or R/W
mov [r8], r12d
shr r12, 32
mov [r8+4], r12d

pop r12
pop r11
pop r10
pop r9
pop rdx
pop rdi
pop r8
pop rbx
pop rax
ret
```

В заключительной части функции (листинг 4.4е) заполняется элемент таблицы страниц, физический адрес был сохранён в самом начале в регистр R12. После выполнения данной процедуры все необходимые структуры для виртуальной страницы будут заполнены, и можно будет обращаться к этой виртуальной странице. Также следует позаботиться о том, чтобы была представлена область виртуальной памяти, где будут находиться таблицы. Об этом желательно позаботиться ещё в защищённом режиме.

Аналогично протекает процедура создания виртуальной страницы размером 2 Мб, за исключением того что в каталоге страниц ещё выставляется бит PS.

Самое время вспомнить фрагмент программы, который мы пропустили в разделе 4.1 (см. листинг 4.5).

Листинг 4.5. Заполнение структур для создания виртуальной страницы

```
mov dword [PDE_addr], 010000011b;PS or Present or Write
mov dword [PDE_addr+4], 0
mov dword [PDPE_addr], PDE_addr or 3 ; Present or Write
mov dword [PDPE_addr+4], 0
mov dword [PML4_addr], PDPE_addr or 3; Present or Write
mov dword [PML4_addr+4], 0
```

В этих строках кода мы заполняем структуры, для того чтобы были представлены первые 2 Мб виртуальной памяти.

Теперь можно проверить правильность написания вышеприведённых процедур. Чтобы убедиться, что функция создания виртуальной страницы работает корректно, выведем три сообщения по разным виртуальным адресам, которые проецированы на один физический адрес.

Листинг 4.6а. Демонстрация работы механизма трансляции адресов в long mode

```
PLM4_BASE_ADDRESS equ 100000h

LM_CODE_START:
   mov rdi, PLM4_BASE_ADDRESS
   call ZeroPage_4K

   xor rax, rax
   mov eax, LM_CODE_START
   and eax, 0FFFFF000h
   mov rbx, rax
   call AllocPage_4K

   mov eax, 0B8000h
   mov rbx, rax
   call AllocPage_4K

   mov eax, 008000h
   mov rbx, rax
   call AllocPage_4K

   mov eax, 07000h
   mov rbx, rax
   call AllocPage_4K
```

В самом начале выделяются страница для кода, страница текстового видеобуфера, и страницы для стека (07000h и 08000h) проецируются на свои оригинальные адреса.

Листинг 4.66. Демонстрация работы механизма трансляции адресов в long mode
(продолжение)

```
    mov eax, 0B8000h
    mov ebx, 12345000h
    call AllocPage_4K

    xor eax, eax
    mov ebx, 0ABCB8000h
    and ebx, 0FFE00000h
    call AllocPage_2M

    mov eax, 0B8000h
    mov rbx, 48A98765000h
    call AllocPage_4K

    xor rax, rax
    mov eax, PLM4_BASE_ADDRESS
    mov cr3, rax

    mov rsi, message
    mov al, 0
    mov ah, 0
    mov bl, "5"
    call OutText

    mov rsi, testmsg1
    mov al, 0
    mov ah, 1
    mov bl, "5"
    mov rdx, 12345000h
    call OutTextEx

    mov rsi, testmsg2
    mov al, 0
    mov ah, 2
    mov bl, "5"
    mov rdx, 0ABCB8000h
    call OutTextEx

    mov rsi, testmsg3
    mov al, 0
    mov ah, 3
    mov bl, "5"
    mov rdx, 48A98765000h
    call OutTextEx

    jmp $

message db "We in long mode!",0
testmsg1 db "Write to address 0x 12345000",0
testmsg2 db "Write to address 0x ABCB8000",0

testmsg3 db "Write to address 0x 0000048A 98765000",0
```

В вышеприведённом коде (листинг 4.6б) для адреса 0ABCB8000h выделяется страница размером 2 Мб (функция `AllocPage_2M`); сделано это в целях разнообразия и для наглядного подтверждения того, что страницы размером 4 Кб и 2 Мб могут использоваться совместно.

Функция `OutText` предназначена для вывода сообщения по адресу 0B8000h; функция `OutTextEx` предназначена для вывода сообщения по адресу, указанному нами в регистре RDX.

Полный исходный код программы находится на компакт-диске в папке part4.

4.2.8. Резюме

В этом разделе были изложены основные принципы работы с виртуальной памятью в 64-битном режиме. В следующем разделе книги речь пойдёт о прерываниях в защищённом режиме.

4.3. Прерывания в long mode

Прерывания являются важнейшей частью любой операционной системы: именно обработка прерываний и исключений позволяет операционной системе организовать взаимодействие с внешними устройствами и защиту своих данных. Поскольку механизм трансляции адресов является одним из условий перехода в long mode, работу с виртуальной памятью в long mode мы рассмотрели раньше, чем работу с прерываниями.

Механизм прерываний в 64-битном режиме почти не изменился по сравнению с защищённым режимом – он даже немного упростился, например, за счёт того, что в long mode не поддерживаются шлюзы задач, т. к. перестал поддерживаться механизм переключения задач.

Также в long mode введён механизм IST. Он позволяет использовать до семи различных фреймов стека для одного обработчика прерываний.

Назначение векторов исключений процессора не изменилось вообще.

4.3.1. Дескрипторы шлюзов

В 64-битном режиме есть только два типа шлюзов: шлюзы прерываний и шлюзы ловушек. Все типы шлюзов расширены до 128 бит для возможности хранения 64-битных адресов обработчиков. Далее на рис. 4.12 приведён формат дескрипторов прерывания и ловушки.

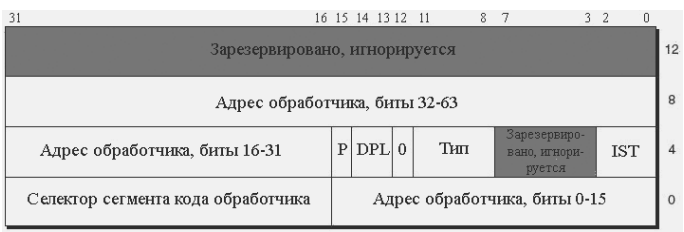


Рис. 4.12. Формат дескриптора шлюза прерывания и ловушки в long mode

В дескрипторе шлюза прерывания поле типа содержит значение 1110b; в дескрипторе шлюза ловушки поле типа содержит значение 1111b. Отличие дескриптора шлюза ловушки от дескриптора шлюза прерывания такое же, как и в защищённом режиме: при переходе через шлюз ловушки процессор не изменяет флаг IF в регистре флагов, а при проходе через шлюз прерывания – сбрасывает флаг IF. Таким образом, обработчик шлюза ловушки прозрачен для прерываний.

Поле IST содержит номер стекового фрейма, используемого обработчиком прерывания. Работа механизма IST будет описана ниже.

4.3.2. Таблица IDT, 64-битный TSS и механизм IST

Как и в защищённом режиме, в long mode все дескрипторы шлюзов объединяются в таблицу IDT. Таблица IDT может находиться по любому адресу; адрес и лимит таблицы прописывается в регистре IDTR. Формат регистра IDTR почти не изменился за исключением того, что поле адреса таблицы расширилось до 64 бит. Как и в защищённом режиме, регистр IDTR загружается/сохраняется с помощью команд LIDT/SIDT. Работа с этими командами ведётся точно так же, как в защищённом режиме, за исключением того что размер операнда надо указать tbyte.

Несмотря на то что многозадачность не поддерживается в long mode, для обеспечения возможности переключения стека при переходе на другие уровни привилегий необходимо создать как минимум один TSS, в котором будут объявлены указатели на стеки для каждого уровня привилегий. Поскольку многозадачность не поддерживается аппаратно в 64-битном режиме, назначение полей, куда сохранялись регистры общего назначения, изменилось – теперь они используются для хранения IST-указателей. На рис. 4.13 приведена схема 64-битного TSS.

Как видно по рисунку, в 64-битном TSS, в отличие от защищённого режима, не хранятся значения сегментных регистров. Кроме того, в 64-битном TSS хранятся 64-битные значения IST1–IST7.

Дескриптор TSS объявляется в GDT и расширен до 128 бит для возможности хранения 64-битного адреса и 20-битного лимита. Формат дескриптора 64-битного TSS приведён на рис. 4.14.

В поле типа для описания должно быть значение 1001b для свободной TSS или значение 1011b для занятой TSS. Для загрузки селектора TSS в регистр TR, как и в защищённом режиме, надо использовать инструкцию LTR; в качестве операнда указан селектор дескриптора TSS в GDT. Дескриптор TSS должен быть помечен как свободный (тип 1001b), после загрузки он автоматически помечается как занятый.

В long mode введён механизм IST (Interrupt Stack Table), позволяющий для обработчика шлюза прерывания и ловушки использовать до 7 различных стеков. В дескрипторах шлюзов в long mode введено новое поле под названием IST. При переходе через шлюз процессор анализирует значение в дескрипторе: если оно не равно нулю, то новый указатель на стек извлекается не из полей RSP0–RSP2, а из полей IST1–IST7 – в зависимости от того, какое значение было указано в поле IST в дескрипторе. Таким образом, для некоторых важных прерываний операционная система может указать надёжный стек (который, к примеру, будет обособлен от общего и всегда будет присутствовать в физической памяти).

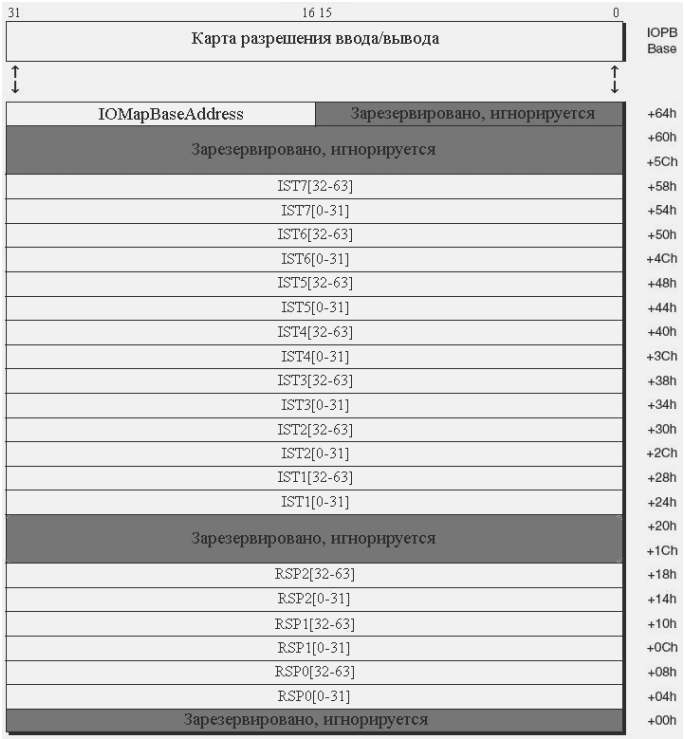


Рис. 4.13. Формат 64-битного TSS

4.3.3. Вызов обработчика прерывания

При вызове обработчиков прерываний основное отличие от защищённого режима заключается в том, что указатель на стек (SS:RSP) помещается в стек обработчика независимо от того, произошёл ли переход на другой уровень привилегий. Код обработчика прерываний обязательно должен быть 64-битным. При смене уровня привилегий в регистр SS помещается нулевое значение (связано это с тем, что в TSS нет полей, содержащих селектор для стека).

Итак, при вызове или генерации прерывания происходит следующее:

1. Анализируется значение поля IST в дескрипторе шлюза: если оно не равно нулю, то загружается соответствующий указатель стека.

Рис. 4.14. Формат системного дескриптора в long mode

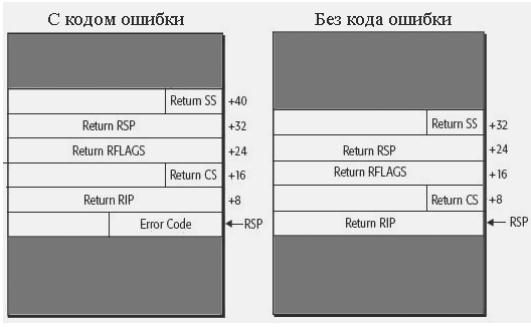


Рис. 4.15. Содержимое стека обработчика прерывания

2. Если поле IST равно нулю и произошло изменение уровня привилегий, загружается указатель на стек для соответствующего уровня привилегий.
3. Полученный указатель стека выравнивается на границу 8 байт (сбрасываются 4 младших бита).
4. Если уровень привилегий изменился, то в регистр SS загружается нулевое значение.
5. В новый стек помещаются предыдущие значения регистров SS и RSP. Регистр SS расширяется до 8 байт.
6. Следующим в стек помещается значение регистра RFLAGS.
7. В регистре RFLAGS очищаются значения TF, NT и RF.
8. Если переход произошёл через шлюз прерывания, то очищается флаг IF в регистре флагов; если переход произошёл через шлюз ловушки, то значение флага IF не модифицируется.
9. В стек обработчика помещаются значения CS и RIP. Значение регистра CS расширяется до 8 байт.
10. Если вектор прерывания подразумевает наличие код ошибки, то в стек помещается код ошибки, предварительно расширенный до 8 байт.
11. Происходит переход на процедуру-обработчик, указатель на которую содержится в дескрипторе шлюза.

Далее на рис. 4.15 приведёно содержимое стека обработчика прерывания (с кодом ошибки и без кода ошибки).

Главное нововведение в long mode при передаче управления обработчику прерывания – это помещение в регистр SS нулевого селектора. Если вызов обработчика произошёл несколько раз, команда IRETQ может извлечь из стека нулевое значение и поместить его в регистр SS без генерации исключения общей защиты – в случае, когда целевой сегмент кода 64-битный и когда целевой уровень привилегий меньше 3.

4.3.4. Практика

Итак, мы изучили механизмы прерываний в long mode. Пора написать программу для закрепления пройденного материала.

Для упрощения программирования напишем пару макросов для объявления шлюзов. Макрос, представленный в листинге 4.7, объявляет 64-битный шлюз. В

качестве параметров принимаются селектор сегмента кода, смещение точки входа, тип шлюза, DPL и IST.

Листинг 4.7. Макрос для объявления дескриптора шлюза

```
macro DEFINE_GATE64 selector, offset, IST, gate_type, DPL
{
   dd (offset and 0FFFFh) or (selector shl 16)
   dd (8000h or (DPL shl 13) or (gate_type shl 8) or IST) or ((offset shr
16) and 0FFFFh)
   dd (offset shr 32)
   dd 0
}
```

В листинге 4.8 представлены два макроса, которые объявляют шлюз прерывания и шлюз ловушки. Оба этих макроса используют макрос, представленный в листинге 4.7 – они создают дескриптор с DPL равным нулю.

Листинг 4.8. Макросы для объявления дескрипторов шлюза прерывания и шлюза ловушки

```
macro DEFINE_INTGATE64 selector, offset, IST
{
   DEFINE_GATE64 selector, offset, IST, INTGATE64, 0
}

macro DEFINE_TRAPGATE64 selector, offset, IST
{
   DEFINE_GATE64 selector, offset, IST, TRAPGATE64, 0
}
```

Константы INTGATE64 и TRAPGATE64 содержат соответствующие значения типов: 1110b и 1111b. Следующий макрос (листинг 4.9) объявляет дескриптор TSS.

Листинг 4.9. Макрос для объявления 64-разрядного TSS

```
macro DEFINE_TSS64Descr BaseAddress, Limit
{
   dd (Limit and 0FFFFh) or ((BaseAddress and 0FFFFh) shl 16)
   dd ((BaseAddress shr 16) and 0FFh) or (TSS64_type shl 8) or (8 shl 12)
or (Limit and 0F0000h) or (BaseAddress and 0FF000000h)
   dd (BaseAddress shr 32)
   dd 0
}
```

Таким образом, объявление таблицы IDT сводится к вызовам макроса DEFINE_INTGATE64 (листинг 4.10).

Листинг 4.10. Объявление IDT

```
IDT64:
  dq 0,0 ;
......;первые 32 вектора зарезервированы под исключения
```

```
  dq 0,0 ; Reserved

  DEFINE_INTGATE64 CODE64_SELEKTOR,IRQ0_handler,0 ; 20 IRQ-0: Timer
  DEFINE_INTGATE64 CODE64_SELEKTOR,int_EOI,0 ; 21 IRQ-1
.........
  DEFINE_INTGATE64 CODE64_SELEKTOR,int_EOI,0 ; 2F IRQ-F

IDT64_Size equ $-IDT64

label IDTR64
    dw IDT64_Size-1
    dq IDT64
```

Метка `IDTR64` указывает на образ регистра IDTR; её можно использовать с командой `LIDT`. Метки `IRQ0_handler` и `int_EOI` указывают на обработчики (о них чуть позже).

Теперь надо объявить TSS для успешного вызова обработчиков прерываний. Структура, представляющая TSS в long mode, стала более чем элементарной (листинг 4.11).

Листинг 4.11. Структура 64-битного TSS

```
struc TSS64
{
  .TSSBase:
  dd ?
  .RSP0 dq ?
  .RSP1 dq ?
  .RSP2 dq ?
  dq ?
  .IST1 dq ?
  .IST2 dq ?
  .IST3 dq ?
  .IST4 dq ?
  .IST5 dq ?
  .IST6 dq ?
  .IST7 dq ?
  dq ?
  dw ?
  .IOMapBase dw $-.TSSBase
}
```

Теперь объявление TSS не сложнее, чем объявление обычной переменной.

```
TSS TSS64
TSS_Size equ $ - TSS
```

А сейчас необходимо объявить новую GDT, содержащую дескриптор нашего TSS (листинг 4.12).

Листинг 4.12. Глобальная дескрипторная таблица

```
GDT64:
   dq 0
   db   0FFh,0FFh,00h,00h,00h,10011010b,11001111b,00h
   db   0FFh,0FFh,00h,00h,00h,10010010b,11001111b,00h
   CODE64Descr   db       00h, 00h,00h,00h, 00h,10011000b,00100000b,00h
   DEFINE_TSS64Descr TSS, TSS_Size-1
GDT64_size     equ        $-GDT64

label GDTR64
     dw GDT64_size-1
     dq GDT64
```

Итак, необходимые структуры объявлены, пора приступить к написанию кода (листинг 4.13).

Листинг 4.13. Инициализация памяти

```
LM_CODE_START:
   mov rdi, PLM4_BASE_ADDRESS
   call ZeroPage_4K

   xor rax, rax
   mov rbx, rax
   call AllocPage_2M

   mov eax, 0200000h
   mov rbx, rax
   call AllocPage_2M

   mov eax, 0400000h
   mov rbx, rax
   call AllocPage_2M

   xor rax, rax
   mov eax, PLM4_BASE_ADDRESS
   mov cr3, rax

   mov rsi, message1
   mov al, 0
   mov ah, 0
   mov bl, "5"
   call OutText
```

В начале происходит создание и заполнение необходимых таблиц и структур для первых 6 Мб памяти. Функция `AllocPage_2M` взята из примера из предыдущей главы. После чего происходит перезагрузка регистра CR3 и вывод первого сообщения.

Листинг 4.14. Инициализация глобальных таблиц

```
lidt tbyte [IDTR64]
lgdt tbyte [GDTR64]

mov [TSS.RSP0], RING0_Stack
mov ax,TSS_SELEKTOR
ltr ax

mov rsi, message2
mov al, 0
mov ah, 1
mov bl, "5"
call OutText
```

В листинге 4.14 производится загрузка регистра IDTR и перезагрузка регистра GDTR. После перезагрузки регистра GDTR не помешала бы перезагрузка сегментных регистров, но мы не будем этого делать. После загрузки регистров IDTR и GDTR происходит заполнение поля `RSP0` в TSS и загрузка регистра TR.

Листинг 4.15. Инициализация контроллера прерывания для последующей работы с прерываниями

```
mov  bx, 2820h

mov  al, 00010001b
out  020h, al
out  0A0h, al
mov  al, bl
out  021h, al
mov  al, bh
out  0A1h, al
mov  al, 00000100b
out  021h, al
mov  al, 2
out  0A1h, al
mov  al, 00000001b
out  021h, al
out  0A1h, al

mov rsi, message3
mov al, 0
mov ah, 2
mov bl, "5"
call OutText

in   al, 70h
and  al, 7Fh
out  70h, al
sti
```

```
    mov rsi, message4
    mov al, 0
    mov ah, 3
    mov bl, "5"
    call OutText

    jmp $
```

В конце мы программируем контроллеры для перенаправления векторов внешних прерываний на векторы 20h–2Fh и разрешаем прерывания.

Теперь необходимо написать обработчики прерываний. Итак, в IDT у нас указан обработчик `IRQ0_handler` для IRQ0 (таймер) и `int_EOI` для всех остальных аппаратных прерываний. В обработчике IRQ0 мы будем выводить количество прошедших миллисекунд после включения прерываний.

Листинг 4.16. Обработчик прерывания на линии IRQ0

```
IRQ0_handler:
    push rax
    push rdx
    push rbx

    inc [counter]

    xor edx, edx
    xor eax, eax
    inc dword [counter]
    mov eax, dword [counter]
    mov ebx, 18;
    div ebx
    cmp edx, 0
    jnz .cont

    inc  byte [sec_counter+3]

    cmp  byte [sec_counter+3],":"
    jnz .cont
    mov byte [sec_counter+3],"0"
    inc byte [sec_counter+2]

    cmp byte [sec_counter+2], ":"
    jnz .cont
    mov byte [sec_counter+2],"0"
    inc byte [sec_counter+1]

    cmp byte [sec_counter+1], ":"
    jnz .cont
    mov byte [sec_counter+1],"0"
    inc byte [sec_counter]
```

```
    cmp byte [sec_counter], ":"
    jnz .cont
    mov byte [sec_counter], "0"
    mov byte [sec_counter+1], "0"
    mov byte [sec_counter+2], "0"
    mov byte [sec_counter+3], "0"

  .cont:
    mov rsi, sec_counter
    mov al, 0
    mov ah, 4
    mov bl, "5"
    call OutText

    pop rbx
    pop rdx
    pop rax

    jmp  int_EOI

int_EOI:
    push rax
    mov  al, 20h
    out  020h, al
    out  0a0h, al
    pop  rax
    iretq
```

Обработчик `int_EOI` является общим для всех обработчиков – он уведомляет оба контроллера прерываний о том, что завершилась обработка прерывания.

Полный исходный код примера находится на компакт-диске, прилагающемся к книге, в папке part4.

4.3.5. Резюме

В этом разделе нами были изучены прерывания в 64-битном режиме процессора. Мы рассмотрели форматы дескрипторов шлюзов, таблицы прерываний и TSS. Изучение 64-битного режима процессора подходит к концу; ниже пойдет речь о механизмах защиты и других аспектах, которые не были изложены в разделах 4.1–4.3.

4.4. Защита и многозадачность

В предыдущих разделах главы 4 шла речь о системе прерываний и работе с виртуальной памятью в 64-битном режиме процессора. В заключительном разделе, посвящённом работе в long mode, будет идти речь о механизмах защиты, многозадачности и других механизмах, которые ещё не обсуждались выше.

В основном защитные механизмы 64-битного режима по сравнению с защищённым режимом не изменились. Здесь будет рассказано об имеющихся отличиях от защищённого режима.

Вообще-то слово «многозадачность» в названии этого раздела совершенно лишнее, поскольку о многозадачности как таковой мы говорить не будем.

Кроме прочего, в этом разделе будут описаны нововведения в long mode, не упоминавшиеся ранее.

4.4.1. Сегменты

Как уже не раз было сказано, в 64-битном режиме не осуществляются проверки лимитов сегментов. Сегменты кода и данных начинаются с нулевого адреса и покрывают всю память. При загрузке селектора в сегментный регистр проверяются параметры защиты, и только после успешной проверки селектор загружается в сегментный регистр. В некоторых случаях, например, при вызове обработчика прерывания, позволяется загрузка в регистр SS нулевого селектора.

В защищённом режиме разрешалось загружать в регистры DS, ES, FS, GS нулевые селекторы, но при любой попытке обратиться через них к памяти генерировалось исключение общей защиты (#GP). В long mode проверка производится только при загрузке селектора в сегментный регистр. Случай, когда в регистр SS при вызове обработчика прерывания загружается нулевой селектор, является исключением, т. к. это делает сам процессор.

Несмотря на то что в 64-битном режиме все сегменты начинаются с нулевого адреса и покрывают всю память, имеется возможность установить начальный адрес для сегментов, описываемых регистрами FS и GS. Процессор предоставляет доступ к теневой части регистров FS и GS через регистры MSR, чтобы обеспечить возможность установки базового адреса для них. Базовый адрес для регистра FS задаёт MSR-регистр под названием IA32_FS_BASE; он имеет индекс C0000100h. Базовый адрес для регистра GS задаёт MSR-регистр под названием IA32_GS_BASE; он имеет индекс C0000101h.

Дескриптор LDT расширился до 16 байт для возможности хранения 64-битного базового адреса LDT. Формат системного дескриптора приводился в предыдущем разделе, на рис. 4.14. Для создания дескриптора LDT необходимо указать значение 0010b в поле типа дескриптора. Из-за того что использование сегментов кода и данных сведено к нулю, польза от использования LDT тоже почти нулевая.

4.4.2. Шлюзы вызова

В long mode шлюзы вызова претерпели небольшие изменения. Дескриптор шлюза расширен до 16 байт для возможности хранения 64-битного адреса обработчика. Теперь в дескрипторе теперь нет поля, указывающего количество параметров для обработчика шлюза. Формат дескриптора шлюза приведён на рис. 4.16. В поле типа должно находиться значение 1100b.

Код обработчика шлюза вызова, как и код обработчика прерывания, должен быть обязательно 64-битным, т. е. в дескрипторе, описывающем код обработчика шлюза, бит L должен быть выставлен.

Действия, производимые процессором, в целом идентичны его поведению при вызове обработчика прерывания. В отличие от защищённого режима, в long mode не копируются параметры из стека, вызывающего в стек обработчика. Сразу воз-

Рис. 4.16. Формат шлюза вызова в long mode

никает вопрос: как же передать параметры обработчику шлюза вызова? Параметры или указатель на параметры (в случае, когда параметров много) можно передать через регистры общего назначения (поскольку их теперь много, это не проблема).

Шлюзы вызова являются основным средством общения 32-битных программ с операционной системой, т. к. предоставляют наиболее простой интерфейс взаимодействия программ, работающих в режиме совместимости с сервисами операционной системы. Инструкции SYSENTER/SYSEXIT не поддерживаются процессорами AMD в long mode (даже в режиме совместимости), а процессоры Intel не поддерживают инструкции SYSCALL/SYSRET в режиме совместимости, поэтому шлюзы вызова – наиболее подходящий метод вызова сервисов 64-битной операционной системы из программ, работающих в режиме совместимости.

4.4.3. Инструкции SYSCALL и SYSRET

Для быстрого вызова системных функций в защищённом режиме существовали инструкции SYSENTER и SYSEXIT. Они позволяли вызывать системные сервисы с минимумом обращений к памяти, а следовательно, с минимальными затратами времени. В защищённом режиме эти инструкции поддерживаются на процессорах обоих компаний. В long mode инструкции SYSENTER и SYSEXIT поддерживаются на процессорах Intel, а на процессорах AMD – нет.

В long mode введены две новые инструкции для быстрого вызова системных процедур: SYSCALL – для вызова системной процедуры и SYSRET – для выхода из системной процедуры. Как и команды SYSENTER/SYSEXIT, инструкции SYSCALL/SYSRET предназначены только для вызова функций и процедур нулевого уровня привилегий из третьего уровня привилегий.

С поддержкой инструкций SYSCALL/SYSRET ситуация полностью противоположная: на процессорах AMD они поддерживаются в защищённом режиме, в режиме совместимости и в 64-битном режиме, но на процессорах Intel – только в long mode (включая режим совместимости). Далее речь пойдёт о том, как эти команды работают на процессорах компании Intel.

Инструкция SYSCALL сохраняет текущее значение регистра RIP в регистре RCX и загружает в регистр RIP значение из MSR-регистра IA32_LSTAR. Также инструкция SYSCALL сохраняет содержимое регистра флагов (младшие 32 бита) в регистр R11 и производит с регистром флагов операцию AND со значением из IA32_FMASK. Таким образом, можно заранее задать маску регистра флагов для обработчика. Селектор целевого сегмента кода загружается из MSR-регистра

IA32_STAR из битов 32–47 (IA32_STAR[47 : 32]). В теневой части регистра CS обнуляется поле DPL; таким образом, целевой код будет выполняться на нулевом уровне привилегий вне зависимости от того, какой сегмент кода описан в GDT (или LDT) по этому селектору. Тем не менее для исключения конфликтных ситуаций операционная система должна позаботиться о недопущении ситуаций, когда дескрипторы в теневой части регистра CS и в таблице (GDT или LDT) будут различаться. В регистр SS заносится селектор, который получается путём добавления числа 8 к селектору сегмента кода.

Инструкция SYSRET заносит в регистр RIP значение из регистра RCX и заносит в регистр флагов (в младшие 32 бита) содержимое регистра R11. Если возврат происходит в 64-битный режим, то происходит следующее: из MSR-регистра IA32_STAR из битов 48–63 (далее IA32_STAR[63 : 48]) извлекается значение и к нему прибавляется 16; полученный селектор загружается в регистр CS, а в регистр SS помещается селектор, полученный аналогичным образом – только вычисленный по формуле IA32_STAR[63 : 48] + 8. Если возврат происходит в режим совместимости, то в регистры CS и SS загружаются следующие селекторы: в CS – IA32_STAR[63 : 48], а в SS – IA32_STAR[63 : 48] + 8. В теневые части регистров SS и CS заносятся такие значения, чтобы в результате процессор находился на третьем уровне привилегий независимо от того, какие дескрипторы описаны в GDT по этим селекторам. Как процессор «узнает», в какой режим происходит возврат? Если размер операнда равен 64 битам, а именно указан префикс REX.W, то возврат происходит в 64-битный режим; если указан размер операнда равный 32 битам (никакого префикса нет), то возврат происходит в режим совместимости.

Регистры IA32_STAR, IA32_LSTAR и IA32_FMASK имеют индексы C0000081h, C0000082h и C0000084h соответственно. Все селекторы, считываемые из регистра IA32_STAR из битов 47–32 и 63–48, должны быть не нулевыми, иначе будет сгенерировано исключение общей защиты.

Для того чтобы узнать, поддерживает ли процессор инструкции SYSCALL/SYSRET, надо вызвать инструкцию CPUID с индексом 80000001h. Если двадцать девятый бит в регистре EDX равен единице, то процессор поддерживает эти инструкции. Проверять это имеет смысл только в защищённом режиме, т. к. если мы перешли в long mode, значит, процессор уже по определению должен поддерживать эти инструкции.

Для того чтобы включить возможность использования команд SYSCALL/SYSRET (по умолчанию их использовать нельзя), необходимо выставить самый первый бит в MSR-регистре IA32_EFER.

4.4.4. Многозадачность

Как уже не раз было сказано, в long mode многозадачность не поддерживается аппаратно. Поэтому реализовывать её разработчикам 64-битных операционных систем придётся уже программно. Впрочем, даже в самой распространённой системе защищённого режима Windows вообще не используется аппаратный механизм многозадачности – в системах Windows он реализуется программно. Возможно, именно этим и руководствовались разработчики компании AMD при разработке

своих 64-разрядных процессоров, ибо зачем реализовывать механизм и нагружать процессор «лишним» функционалом, если им всё равно никто не пользуется?

Как следствие отсутствия аппаратной поддержки многозадачности, не поддерживаются: дальний прыжок (JMP/CALL), возврат из прерывания (IRETQ) и вообще любые команды, в которых в качестве целевого сегмента (сегмента возврата) указан селектор дескриптора TSS. Единственный способ загрузить значение в регистр TR – это команда LTR, т. к. она не инициирует механизм переключения с одной задачи на другую.

Для возможности переключений стека и запрещения использования некоторых портов ввода/вывода по-прежнему необходимо держать в памяти как минимум один TSS. Как и в защищённом режиме, глобальной дескрипторной таблице необходимо объявить дескриптор этого TSS и в регистр TR загрузить его селектор.

Итак, вспомним формат 64-битного TSS (рис. 4.17). Для запрещения некоторых (или всех) портов используется карта разрешения ввода/вывода. Поле `IOMapBaseAddress` содержит смещение карты ввода/вывода относительно начала TSS. Формат карты разрешения ввода/вывода не изменился по сравнению с защищённым режимом. Контроль ввода/вывода тоже не изменился (см. раздел 2.4).

Для реализации программной многозадачности необходимо будет завести для каждой задачи свою область памяти, в которой будет храниться информация о

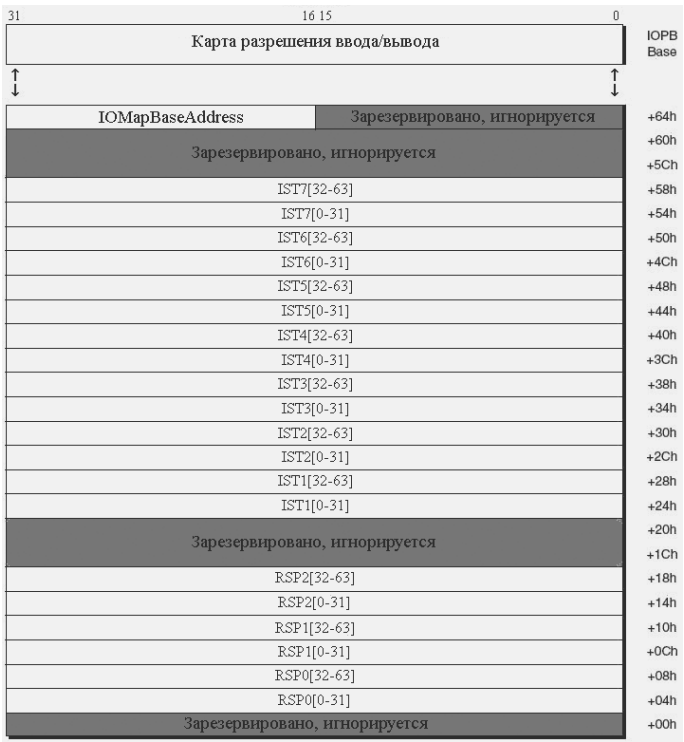


Рис. 4.17. Формат 64-битного TSS

задаче: состояние регистров общего назначения, регистра флагов и других параметров. Для переключения задач необходимо будет вручную сохранить состояние текущей задачи, загрузить состояние другой задачи и передать управление на место, где она была прервана.

4.4.5. Практика

В качестве практического примера напишем программу, которая будет использовать инструкции SYSCALL/SYSRET.

Поскольку в примере код будет выполняться на нулевом и третьем уровне привилегий, надо изменить код функции, которая создаёт структуры, описывающие виртуальные страницы. Бит U/S должен быть выставлен в каждой структуре, описывающей искомую страницу. Изменённый код функции `AllocPage_2M` приводиться здесь не будет – его можно найти на компакт-диске, прилагающемся к книге – в исходнике, относящемся к данному разделу.

Итак, после выделения необходимой памяти нам надо обновить GDT. Из-за особенностей работы инструкций SYSCALL/SYSRET GDT должна быть построена особым образом (листинг 4.17).

Листинг 4.17. Глобальная дескрипторная таблица

```
GDT64:
       dq 0
       db            0FFh, 0FFh, 00h, 00h, 00h, 10011010b, 11001111b,
00h                  ; 08
       db            0FFh, 0FFh,00h,00h,00h, 10010010b, 11001111b,00h
                     ; 10

CODE64R0_Descr   db   00h, 00h,00h,00h, 00h, 10011000b, 00100000b,00h ;18
DATA64R0_Descr   db   0FFh, 0FFh, 00h, 00h, 00h, 10010010b, 11001111b,00h
                     ;20

DATA64R3_Descr   db   0FFh, 0FFh,  00h, 00h, 00h, 11110010b, 11001111b,
00h                  ;28
CODE64R3_Descr   db   00h, 00h,00h,00h, 00h, 11111000b, 00100000b,00h
                     ;30

GDT64_size       equ           $-GDT64

label GDTR64
      dw GDT64_size-1
      dq GDT64
```

Первые два дескриптора – это дескрипторы, оставшиеся нам от защищённого режима (ничего не будем с ними делать); потом идут два дескриптора для нулевого кольца и два дескриптора – для третьего.

Следующее, что нам надо сделать – это выставить нулевой бит в MSR-регистре IA32_EFER, для того чтобы разрешить выполнение инструкций SYSCALL/SYSRET. После чего необходимо поместить базовые селекторы в регистр IA32_STAR,

занести указатель на код нулевого кольца в регистр IA32_LSTAR и занести маску регистра флагов для нулевого кольца в регистр IA32_FMASK. В листинге 4.18 приведён код, который включает механизм SYSCALL/SYSRET.

Листинг 4.18. Инициализация механизма SYSCALL/SYSRET

```
mov ecx, IA32_EFER
rdmsr
or eax,1
wrmsr

pushf
xor r11, r11
pop r11

mov ecx, IA32_STAR
xor eax, eax
mov edx, 000230018h
wrmsr

mov ecx, IA32_LSTAR
mov eax, RING0_CODE
xor edx, edx
wrmsr

mov ecx, IA32_FMASK
mov eax, r11d
xor edx, edx
wrmsr

mov rsi, message3
mov al, 0
mov ah, 2
mov bl, "5"
call OutText

xor rcx, rcx
mov ecx, RING3_CODE
;sysretq
db 48h, 0Fh, 07h
```

Нельзя забывать, что в регистре IA32_STAR в старших 16 битах должен находиться базовый селектор для третьего кольца, и поле RPL у него должно быть равно 3. В конце вышеприведённого кода мы переходим на третье кольцо – для этого заносим в регистр RCX указатель на код третьего кольца и вызываем инструкцию SYSRET. Команда SYSRET задана своим опкодом – 48h, префикс REX.W, а опкод инструкции SYSRET – 0F07h. Это нужно для стопроцентной гарантии того, что мы вернёмся в 64-битный режим, а не в режим совместимости.

Теперь осталось только привести сам код третьего и нулевого кольца (листинг 4.19).

Листинг 4.19. Код третьего кольца и обработчика SYSCALL

```
RING3_CODE:
     mov ax, 02Bh
     mov ds, ax

      @@:
     inc dword [R3_c]
     mov rsi, message4
     mov al, 0
     mov ah, 3
     mov bl, "5"
     call OutText

     syscall
     jmp @b

RING0_CODE:
     mov ax, 20h
     mov ds, ax

     inc dword [R0_c]
     mov rsi, message5
     mov al, 0
     mov ah, 4
     mov bl, "5"
     call OutText

     ;sysretq
     db 48h, 0Fh, 07h

message1 db "We are in long mode!",0
message2 db "GDT and TR are updated",0
message3 db "SYSCALL/SYSRET mechanism was initialized",0
message4 db "RING3 code running   "
   R3_c dd 0
      db 0
message5 db "RING0 code running   "
   R0_c dd 0
      db 0
```

Коды третьего и нулевого кольца просто выводят сообщение и каждый раз перед выводом сообщения обновляют счётчик. Если после запуска программы рядом со строками «RING3 code running» и «RING0 code running» мелькают «закорючки», значит, программа работает правильно.

Программу и её исходный код можно найти на компакт-диске, прилагающемся к книге, в папке part4.

4.4.6. Резюме

В последнем разделе, посвященном описанию работы в 64-битном режиме процессора, мы изучили шлюзы вызова в long mode, а также механизм быстрых системных вызовов с использованием инструкций SYSCALL/SYSRET. Следующая глава будет посвящена программированию на ассемблере в 64-битных системах Windows.

5 ПРОГРАММИРОВАНИЕ В WIN64

5.1. Введение в Win64

В предыдущей главе мы говорили о 64-битном режиме процессора. В главе 3 речь шла о программировании на ассемблере в 32-битных операционных системах семейства Windows. Теперь, после изучения архитектуры работы процессора в 64-битном режиме, настало время изучить программирование в 64-битных системах Windows, или в Win64.

Первые 64-битные версии Windows появились у систем Windows XP и Windows 2003 Server, причем почти одновременно, в 2005 году. Следующие версии операционных систем Windows (2008 Server, Vista и др.) были как 32-битными, так и 64-битными. Вообще 64-битные версии есть только у систем с архитектурой WinNT (по понятным причинам у систем Win9x 64-битных версий нет).

5.1.1. Преимущества и недостатки

Каковы же преимущества использования 64-битных версий Windows? Основное и главное преимущество 64-битных версий – это поддержка большого объёма виртуальной и физической памяти. Наиболее остро эта необходимость ощущается на серверных системах, на которых работают требовательные к количеству памяти приложения (например, СУБД). Как известно, на 32-битных версиях систем приложение может получить доступ к 2 Гб памяти (в некоторых случаях – к 3 Гб). Также есть механизм AWE, который позволяет получить доступ к более чем 2 Гб памяти, но это, мягко говоря, не то, что надо. На 64-битных версиях можно получить доступ к 8 Тб виртуальной памяти (в дальнейшем этот объем может быть увеличен), что более чем достаточно для работы требовательных к памяти серверных приложений. Ещё одно преимущество 64-битных систем – высокая скорость вычислений: во-первых, появился расширенный набор инструкций для работы с числами с плавающей точкой, а во-вторых, работа с числами с двойной точностью (Double и QWORD) стала на порядок быстрее – операции, которые раньше требовали нескольких шагов, теперь проходят за один шаг.

Но у 64-битных систем есть и минусы. Во-первых, все 64-битные команды стали как минимум на 1 байт длиннее, и в итоге исполняемый код занимает больше места в памяти. Во-вторых, из-за большого количества структур для описания виртуальной памяти возникает очень много «дополнительных расходов» памяти, и чем больше мы используем виртуальную память, тем больше места уходит под таблицы PML4, PDPT, PD и PT. При повседневном использовании, даже если просуммировать общее потребление памяти со всеми процессами, оно редко превышает 1 Гб. Для описания 1 Гб памяти в Win64 уйдёт как минимум в два раза

больше физической памяти, чем в Win32. Именно по этой причине 64-битные версии Windows почти не дают преимуществ в повседневном использовании и не очень сильно распространены среди обычных пользователей.

Пока ещё ничего не было сказано про увеличение количества регистров общего назначения. Их стало вдвое больше, и теперь при правильном написании кода программы можно уменьшить количество обращений к памяти, т. к. теперь есть где сохранить промежуточные значения и данные. Но увеличение количества регистров общего назначения плохо влияет на механизм переключения задач: ведь при переключении контекстов потоков надо сохранять и загружать больше регистров и данных.

5.1.2. Память в Win64

Итак, как же устроена виртуальная память в Win64? Отличий от Win32 не так много. По-прежнему каждый процесс находится в своём собственном виртуальном адресном пространстве. По-прежнему память разделена на две части, но уже не пополам. Память третьего кольца находится в диапазоне 0h–80000000000h – в нём недоступны по 64 Кб «сверху» и «снизу», и в итоге у нас имеется 8 Тб минус 128 Кб. Память ядра находится в диапазоне 0000080000000000h–FFFFFFFFFFFFFFFFh. На первый взгляд кажется, что это очень большой диапазон памяти, но если учесть, что адрес в long mode может быть только в канонической форме (фактически используются только 48 бит), то это равносильно диапазону FFFF080000000000h–FFFFFFFFFFFFFFFFh. Старшие 16 бит адреса могут быть как нулями, так и единицами – просто исторически сложилось так, что адреса памяти ядра принято обозначать единицами в старших битах.

Итак, что же у нас имеется в общей сложности? Надо учесть, что виртуальный адрес в long mode всегда должен быть в канонической форме. 8 Тб виртуальной памяти отведено для приложений третьего кольца и 248 ТБ виртуальной памяти – для ядра системы; этого с избытком хватит даже самым требовательным к памяти приложениям и драйверам.

5.1.3. Модель вызова

В отличие от Win32, в Win64 используется модель вызова fastcall. В Win32 использовались три модели вызова функций: cdecl, stdcall и fastcall. Модель вызова – пожалуй, самое значимое отличие Win64 от Win32. Разработчики 64-разрядных операционных систем Windows стандартизировали вызов функций в Win64; теперь все функции, которые предоставляет ОС, можно вызывать только по соглашению fastcall (разумеется, свои собственные функции можно вызывать как вам удобнее).

Вызов API в Win64 стал сложнее, чем вызов fastcall-функций в Win32, и, по сути, от соглашения fastcall осталось только одно название. В Win64 fastcall выглядит так: первые четыре параметра передаются соответственно через RCX, RDX, R8, R9, а остальные – через стек в обратном порядке (как в stdcall). Также всегда резервируется место в стеке под первые четыре параметра (даже если количество параметров меньше четырёх), чтобы потом можно было переместить значения из регистров в стек и использовать их «по старинке» ([ebp+xx]), а сами регистры

использовать для других целей. Результат выполнения функции передаётся через регистр RAX или через XMM0, если он является числом с плавающей точкой.

Место в стеке для первых четырёх параметров, которые передаются через регистры, называется *теневой частью*. Резервирование места достигается тем, что из регистра RSP вычитается значение 32 (стек растёт сверху вниз). За очистку стека от параметров и теневой части отвечает вызывающая процедура. Следует помнить, что стек всегда должен быть выровненным на границу 16 байт, т. е. указатель в RSP должен быть кратен 16. Это требование неочевидное, но тем не менее так нам говорит документация. В большинстве случаев отсутствие выравнивания на 16 байт не повлечет за собой никаких последствий, однако некоторые функции вызовут исключение именно по этой причине. В общем случае отсутствие выравнивания уменьшает производительность приложения.

Также, если в первых четырёх параметрах есть значения с плавающей точкой, то они дублируются через XMM-регистры (XMM0L, XMM1L, XMM2L и XMM3L). Если размер параметра меньше 64 бит, то он расширяется до 64 бит нулями или единицами (если он отрицателен) и только потом помещается в регистр или стек.

Итак, например, у нас есть функция и она принимает шесть параметров: `function(int1,real1,int2,real2,int3,real3)` – три целочисленных параметра и три параметра с плавающей запятой. В соответствии с соглашением вызова в Win64 вызов произойдёт следующим образом:

1. `push real3`
2. `push int3`
3. `R9=real2, XMM3L=real2`
4. `R8=int2`
5. `RDX= real1, XMM1L=real1`
6. `RCX=int1`
7. `RSP= RSP-32`
8. `Вызов функции`
9. `Очистка стека (если это необходимо)`

Обычно резервирование места происходит в самом начале, а параметры помещаются в стек не командами `PUSH`, а командой `MOV`.

Очистка параметров в стеке в Win64 несколько необычна. Как уже было сказано выше, за очистку стека от параметров отвечает вызвавшая функция, а не вызываемая. Таким образом, для оптимизации можно в начале программы или подпрограммы зарезервировать в стеке пространство, достаточное для вызова функции с самым длинным списком параметров. В отличие от процедур и функций в Win32, которые при вызове других функций явно добавляют параметры в стек с помощью инструкций `PUSH`, при анализе кода функций в Win64 редко встретишь команду `PUSH`: вместо неё используются команды типа `mov [esp+n], val`. Команда `MOV`, в отличие от команды `PUSH`, работает быстрее, т. к. она только записывает данные в память, а команда `PUSH` изменяет регистр RSP и записывает данные в память.

С учётом всего вышесказанного при правильном написании программ в Win64 можно минимизировать количество модификаций регистра RSP и количество обращений к стеку; следовательно, увеличивается и скорость вызова процедур и функций.

Тем не менее все эти знания вряд ли нам пригодятся, т. к. в компиляторе FASM уже есть необходимый набор макросов, позволяющий одной строчкой производить вызов функций и максимально облегчающий написание процедур и функций, отвечающих модели вызова в fastcall Win64.

Следующие регистры могут измениться после вызова fastcall функции: RAX, RCX, RDX, R8-R11, XMM0–XMM5. Если значения этих регистров нужны вызывающему, он должен их сохранить. Остальные регистры (RBX, RSI, RDI, RSP, RBP, R12-R15, XMM6–XMM15) не должны измениться после вызова fastcall-функции, и если вызываемая функция их использует, то она должна их сохранить. При написании функций обратного вызова необходимо следовать этим правилам, иначе неизбежны ошибки в работе системных функций.

Правила построения прологов и эпилогов функции не изменились по сравнению с Win32; регистр RBP по-прежнему используется как указатель стекового фрейма функции.

Передача параметров функциям через регистры XMM и работа с числами с плавающей точкой в данном разделе и в данной редакции книги рассматриваться не будет.

5.1.4. Режим совместимости

Для обеспечения обратной совместимости старые Win32-приложения третьего кольца могут работать на Win64 без каких-либо серьёзных ограничений. Но драйверы режима ядра обязательно должны быть 64-битными – следовательно, все антивирусы, файрволы и другие приложения, устанавливающие свои драйверы, тоже должны быть 64-битными.

Все 32-битные приложения выполняются в подсистеме WOW64. При вызове системных сервисов и функций 32-битными программами все вызовы проходят сначала через их 32-битные аналоги и только потом направляются к 64-битным версиям. Старым 32-битным приложениям по-прежнему доступны младшие 2 Гб памяти, но если компилировать программы со специальным флагом, то им могут быть доступны все 4 Гб памяти. Также нельзя комбинировать разные типы кода: 32-битные программы не могут загружать 64-битные DLL, и наоборот. При загрузке некоторой системной DLL, имя которой одинаково в обоих версиях Windows, подсистема WOW64 автоматически перенаправляет запросы на 32-битные версии библиотек.

5.1.5. Win64 API и системные библиотеки

Система DLL библиотек, содержащих API-функции, почти не изменилась по сравнению с Win32. Большинство функций имеют прежние имена и параметры, но размерность зависимых от платформы параметров изменилась (например, все указатели стали 64-битными). Имена и назначение системных библиотек также остались прежними, изменился только код внутри них (следовало бы ожидать, что, например, kernel32.dll переименуют в kernel64.dll – но нет, имя сохранено без изменений).

5.1.6. Практика

Итак, базовые сведения о Win64 мы получили; теперь напишем приложение, выводящее до боли знакомую фразу «Hello world!» в 64-разрядной операционной системе Windows.

Листинг 5.1. Исходный код приложения

```
format PE64 GUI 5.0
entry start

include 'win64a.inc'

section '.data' data readable writeable

   text  db 'Hello world!',0

section '.code' code readable executable

start:
   sub rsp, 32
   mov rcx, 0
   mov rdx, text
   mov r8,  text
   mov r9, 0
   call [MessageBox]

   xor rcx, rcx
   call [ExitProcess]


section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL',\
     user32,'USER32.DLL'

  import kernel32,\
    ExitProcess, 'ExitProcess'

  import user32,\
    MessageBox,'MessageBoxA'
```

Единственное что здесь может быть непонятно с первого взгляда, это команда `sub rsp, 32`. Этого требует соглашение fastcall – резервировать место в стеке для четырёх параметров необходимо в любом случае, даже если параметр меньше 4. Нет необходимости в команде `add rsp, 32`, т. к. функция `ExitProcess` уже никогда не вернёт управление. Нельзя забывать, что в подпрограммах и функциях, которые вызывают другие fastcall-функции, необходимо возвращать указатель стека на своё место, поскольку будет потерян адрес возврата, необходимый команде `RET`.

Как бы выглядел код этого приложения, если бы мы воспользовались макросом invoke?

Листинг 5.2. Исходный код приложения с применением макроса invoke

```
format PE64 GUI 5.0
entry start

include 'win64a.inc'

section '.data' data readable writeable

text  db 'Hello world!',0

section '.code' code readable executable

start:

        invoke  MessageBox,0,text,text,0

        invoke  ExitProcess,0

section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL',\
        user32,'USER32.DLL'

  import kernel32,\
        ExitProcess, 'ExitProcess'

  import user32,\
        MessageBox,'MessageBoxA'
```

Макрос invoke скрывает от нас все тонкости вызова API-функций и максимально облегчает программирование. Благодаря этому макросу изменений в коде программы по сравнению с Win32 почти нет. Главное отличие – это формат исполняемого файла: PE64 (или PE32+); в остальном код такой же.

5.1.7. Резюме

В этом разделе мы познакомились с 64-битными версиями операционных систем Windows и изучили основные отличия от Win32. В следующем разделе будут более подробно изучены некоторые аспекты программирования на ассемблере под Win64.

5.2. Программирование в Win64

В разделе 5.1 было представлено введение в программирование на 64-битных версиях систем Windows. В этом разделе мы более подробно изучим ключевые аспекты программирования в Win64 – как в третьем кольце, так и в нулевом.

Программирование в Win64 и Win32 принципиально ничем не различается. Отличия заметны лишь при низкоуровневом программировании, когда дело касается именно ассемблера. При разработке приложений на языках высокого уровня отличий в программах для Win32 и Win64 почти нет – от программиста требуется только перекомпиляция программы под Win64. Впрочем, вы в этом сами убедитесь после прочтения этого раздела.

5.2.1. Изменения в типах данных

Итак, что же изменилось в типах данных? Изменений очень много; ниже будут приведены только общие сведения о типах данных, а в целях получения более точной информации о типах данных в Win64 следует анализировать заголовочные файлы для программ на языке C/C++, входящих в состав SDK и DDK (WDK). При программировании на ассемблере эти файлы являются основным источником информации о типах данных и их размерах.

Очевидно то, что все указатели теперь имеют размер 8 байт. Это значит, что все типы, названия которых начинаются с буквы `P` (по терминологии Microsoft), теперь имеют размер 8 байт.

Во-вторых, все хендлы теперь имеют размер 8 байт, т. е. все типы, названия которых начинаются с буквы `H`, а именно: `HANDLE`, `HBRUSH`, `HBITMAP`, `HCOLORSPACE`, `HCURSOR`, `HDC`, `HFONT`, `HICON`, `HINSTANCE`, `HKEY`, `HMENU`, `HMODULE`, `HPEN`, `HPALETTE`, `HWND` и т. д.

В-третьих, параметры оконных сообщений `WPARAM` и `LPARAM` тоже теперь имеют размер 8 байт.

Типы данных, названия которых говорят сами за себя, и те типы, размеры которых сложились исторически, не изменились, например: `CHAR` – по-прежнему 1 байт, `UINT`, `UINT32`, `ULONG`, `ULONG32` – по-прежнему 4-байтовые числа без знака, `UINT64`, `ULONG64`, `ULONGLONG` – по-прежнему 8-байтовые числа без знака, `BOOL` – по-прежнему 4 байта и т. д.

Изменения касаются и ядра системы. Поскольку в ядре системы в основном используются указатели, то большинство параметров, принимаемых функциями, теперь имеют размер 8 байт, за исключением тех, о которых было сказано в предыдущем абзаце.

5.2.2. Выравнивание стека

В предыдущей главе вскользь шла речь про 16-байтовое выравнивание стека, но не было сказано, зачем оно нужно. Если стек не выравнивать, то никаких ошибок и исключений не возникнет – тем не менее компания Microsoft требует этого от программ. Так зачем же надо всегда выравнивать указатель стека на 16-байтовую границу?

Причина тому – 128-битные XMM-регистры. Т. к. они имеют размер 16 байт, то и хранить их в памяти лучше по адресу, выровненному на 16 байт. Выравнивание стека на 16 байт делает его удобным для хранения XMM-регистров. Обработчики системных прерываний и системных вызовов сохраняют XMM-регистры в стеке прерванной задачи, и поэтому для максимально быстрого сохранения/

восстановления регистров XMM требуется, чтобы указатель на стек всегда был выровнен на 16 байт.

В примере из раздела 5.1 мы пренебрегли выравниванием стека; тем не менее в любой более сложной программе этим пренебрегать нельзя, иначе возможны потери в производительности как самой программы, так и системы в целом.

5.2.3. GUI-приложения

В качестве практического примера программирования в третьем кольце в Win64 напишем оконную программу, аналогичную той, которая была представлена в разделе 3.2. Далее в листинге 5.3 приведён исходный код оконной программы для Win64.

Листинг 5.3. Создание окна приложения

```
start:
   sub rsp, 8*5
   xor rcx, rcx
   call [GetModuleHandle]

   mov [hInst], rax
   mov [wc.style], CS_HREDRAW + CS_VREDRAW + CS_GLOBALCLASS
   mov rbx, WndProc
   mov [wc.lpfnWndProc],  rbx
   mov [wc.cbClsExtra], 0
   mov [wc.cbWndExtra], 0
   mov [wc.hInstance], rax

   mov rdx, IDI_APPLICATION
   xor rcx, rcx
   call [LoadIcon]
   mov  [wc.hIcon], rax

   mov rdx, IDC_ARROW
   xor rcx, rcx
   call [LoadCursor]
   mov [wc.hCursor], rax

   mov [wc.hbrBackground], COLOR_BACKGROUND+1
   mov qword [wc.lpszMenuName], 0
   mov rbx, szClassName
   mov qword [wc.lpszClassName], rbx

   mov rcx, wc
   call [RegisterClass]

   sub rsp, 8*8

   xor rcx, rcx
   mov rdx, szClassName
   mov r8, szTitleName
```

```
mov r9, WS_OVERLAPPEDWINDOW
mov qword [rsp+8*4], 50
mov qword [rsp+8*5], 50
mov qword [rsp+8*6], 300
mov qword [rsp+8*7], 250
mov qword [rsp+8*8], rcx
mov qword [rsp+8*9], rcx
mov rbx, [hInst]
mov [rsp+8*10], rbx
mov [rsp+8*11], rcx
call [CreateWindowEx]
mov [main_hwnd], rax

xor rcx, rcx
mov rdx, button_class
mov r8, AboutTitle
mov r9, WS_CHILD
mov qword [rsp+8*4], 50
mov qword [rsp+8*5], 50
mov qword [rsp+8*6], 200
mov qword [rsp+8*7], 50
mov rbx, [main_hwnd]
mov qword [rsp+8*8], rbx
mov qword [rsp+8*9], rcx
mov rbx, [hInst]
mov [rsp+8*10], rbx
mov [rsp+8*11], rcx
call [CreateWindowEx]
mov [AboutBtnHandle], rax

xor rcx, rcx
mov rdx, button_class
mov r8, ExitTitle
mov r9, WS_CHILD
mov qword [rsp+8*4], 50
mov qword [rsp+8*5], 150
mov qword [rsp+8*6], 200
mov qword [rsp+8*7], 50
mov rbx, [main_hwnd]
mov qword [rsp+8*8], rbx
mov qword [rsp+8*9], rcx
mov rbx, [hInst]
mov [rsp+8*10], rbx
mov [rsp+8*11], rcx
call [CreateWindowEx]

mov [ExitBtnHandle], rax

add rsp, 8*8
```

```
 mov rdx, SW_SHOWNORMAL
 mov rcx, [main_hwnd]
 call [ShowWindow]

 mov rcx, [main_hwnd]
 call [UpdateWindow]

 mov rdx, SW_SHOWNORMAL
 mov rcx, [AboutBtnHandle]
 call [ShowWindow]

 mov rdx, SW_SHOWNORMAL
 mov rcx, [ExitBtnHandle]
 call [ShowWindow]

msg_loop:
 xor r9, r9
 xor r8, r8
 xor rdx, rdx
 mov rcx, msg
 call  [GetMessage]

 cmp  rax,1
 jb end_loop
 jne  msg_loop

 mov rcx,  msg
 call [TranslateMessage]
 mov rcx,  msg
 call [DispatchMessage]
 jmp  msg_loop

end_loop:
 xor rcx, rcx
 call [ExitProcess]
```

После просмотра программы сразу возникает вопрос: почему не использован макрос invoke, а вместо этого вызов API-функций происходит вручную? Потому что использование макроса invoke порождает много лишних команд `sub rsp, n/ add rsp, n`, между тем как можно сделать это только один раз (или несколько раз, как в нашем примере). Это вовсе не означает, что такой способ «более правильный» – наша задача лишь показать альтернативный способ вызова API-функций.

В коде программы нет абсолютно ничего нового, кроме вызова fastcall-функций. В самом начале командой `sub rsp, 8*5` резервируется место в стеке для четырёх параметров и происходит выравнивание стека на границу 16 байт (32 байта для параметров + 8 байт для выравнивания = 40 байт). Перед использованием функции CreateWindowEx в стеке резервируется дополнительно 64 байта под восемь параметров для этой функции (функция принимает двенадцать параметров; место под четыре параметра уже зарезервировано).

Основной код программы написан – теперь осталось написать саму оконную функцию. Её код приведён в листинге 5.4.

Листинг 5.4. Оконная функция

```
proc WndProc hwnd, wmsg, wparam, lparam
   sub rsp, 8*4
   cmp rdx, WM_DESTROY
   je  .wmdestroy
   cmp rdx, WM_COMMAND
   jne .default
   mov rax, r8
   shr rax, 16
   cmp rax, BN_CLICKED
   jne .default
   cmp r9, [AboutBtnHandle]
   je  .about
   cmp r9, [ExitBtnHandle]
   je  .wmdestroy

.default:
   call [DefWindowProc]
   jmp .finish

.about:
   xor rcx, rcx
   mov rdx, AboutText
   mov r8, AboutTitle
   xor r9, r9
   call [MessageBox]
   jmp .finish

.wmdestroy:
   xor rcx, rcx
   call [ExitProcess]
.finish:
   add rsp, 8*4  ; restore stack
   ret
endp
```

В оконной функции для вызова функций `DefWindowProc` и `MessageBoxA` сразу резервируется в стеке место для четырёх параметров. В начале оконной функции происходит проверка кода сообщения (который содержится в регистре RDX), и, если этот код не равен ни `WM_DESTROY`, ни `WM_COMMAND`, то происходит вызов функции `DefWindowProc`. Притом нет необходимости передавать ей параметры – все параметры уже находятся на своих местах, т. к. функция `DefWindowProc`, по сути, тоже является оконной.

В оконной функции нет необходимости выравнивать указатель стека на 16 байт. Системная функция, которая вызывает оконную, использует выровненный стек;

после вызова в стек помещается адрес возврата, а после этого надо поместить в стек ещё что-то, чтобы он стал выровненным. Почти все fastcall-функции (так же, как и stdcall) – и наша функция здесь не исключение – первой командой сохраняют значение регистра RBP в стеке; таким образом, стек выравнивается на границу 16 байт.

5.2.4. Программирование драйверов

В программировании драйверов для win64 появились дополнительные ограничения в связи с введением в 64-битные версии операционных систем новой технологии для защиты системного кода и данных. В результате в 64-битных версиях операционных систем Windows драйверам режима ядра запрещено модифицировать код всех системных функций и всех критических структур данных (таких как GDT, IDT и т. д.). Любое нарушение этого требования приведёт к появлению «синего экрана смерти» (BSOD).

В качестве практического примера программирования в нулевом кольце в Win64 напишем драйвер, аналогичный тому, который был написан в разделе 3.3, только теперь драйвер будет передавать приложению третьего кольца не глобальную дескрипторную таблицу, а таблицу дескрипторов прерываний.

Листинг 5.5. Основой обработчик пакетов IRP драйвера idtdump

```
proc DispatchControl pDeviceObject, pIrp
local status:DWORD

   push rdi
   push rsi
   push rbx
   sub rsp, 8    ; for align stack

   mov [status], STATUS_DEVICE_CONFIGURATION_ERROR

   virtual at rdx  ; rdx = pIRP
  .rdx IRP
   end virtual

   mov rsi, [.rdx.Tail.Overlay.CurrentStackLocation]
   virtual at rsi
  .rsi IO_STACK_LOCATION
   end virtual

   cmp [.rsi.Parameters.DeviceIoControl.IoControlCode], IOCTL_DUMP_IDT
   jz .IOCTL_DUMP
   jmp .next
  .IOCTL_DUMP:
   sidt tword [IDTR]
   xor rcx, rcx
   mov cx, [IDTR.Limit]
   inc rcx  ; ecx = IDT size
```

```
 xor rax, rax
 xor rbx, rbx
 mov ebx, [.rsi.Parameters.DeviceIoControl.InputBufferLength]
 cmp ebx, 8
 jnz .copydata
 mov rax, [.rdx.AssociatedIrp.SystemBuffer]
 mov ebx, [rax+4]  ; ebx = needed data size
 mov eax, [rax]    ; eax = start offset in GDT

 add rbx, rax
 cmp rbx, rcx
 jna @f
 mov rbx, rcx
@@:
 sub rbx, rax   ; ebx = fact data size
 cmp rbx, 7FFFFFFFh
 jna @f
 mov rbx, 0
@@:
 mov rcx, rbx
.copydata:
 mov [status], STATUS_BUFFER_TOO_SMALL
 cmp ecx, [.rsi.Parameters.DeviceIoControl.OutputBufferLength]
 ja .error

 mov rbx, rcx    ;  save rcx in rbx, rbx = data size
 mov rsi, [IDTR.Address]
 add rsi, rax
 mov rdi, [.rdx.AssociatedIrp.SystemBuffer]
 rep movsb

 mov [.rdx.IoStatus.Information], rbx

 mov [status], STATUS_SUCCESS
 jmp .exit

.next:
 jmp .exit
.error:
 mov [.rdx.IoStatus.Information], 0

.exit:
 mov eax, [status]
 mov  [.rdx.IoStatus.Status], eax

 invoke IofCompleteRequest, rdx, IO_NO_INCREMENT
 mov eax, [status]

 add rsp ,8
```

```
    pop rbx
    pop rsi
    pop rdi
    ret
endp
```

Формат, размер и назначение параметров по сравнению с 32-битной версией драйвера не изменились. Начальное смещение в таблице и размер копируемых данных по-прежнему имеют размер 4 байта, поэтому код, проверяющий их правильность, не изменился. Функция-обработчик запроса `IRP_MJ_CREATE` (обработчик запроса `IRP_MJ_CLOSE` полностью ей идентичен) и функция `DriverEntry` приведены в листинге 5.6.

Листинг 5.6. Обработчик запроса IRP_MJ_CREATE драйвера idtdump и его точка входа

```
proc DispatchCreate pDeviceObject, pIrp

    virtual at rdx
   .rdx IRP
    end virtual

    mov [.rdx.IoStatus.Status], STATUS_SUCCESS
    and [.rdx.IoStatus.Information], 0

    sub rsp, 32
    mov rcx, rdx
    mov rdx, IO_NO_INCREMENT
    call [IofCompleteRequest]
    add rsp, 32

    mov eax, STATUS_SUCCESS
    ret

endp

proc DriverEntry pDriverObject, pusRegistryPath
local status:DWORD

    mov [pDriverObject], rcx
    mov [status], STATUS_DEVICE_CONFIGURATION_ERROR

    invoke IoCreateDevice, [pDriverObject], 0, DeviceName, FILE_DEVICE_
UNKNOWN, 0, FALSE, DeviceObject

    cmp eax, STATUS_SUCCESS
    jnz .END
    invoke IoCreateSymbolicLink, SymbolicLinkName, DeviceName
    cmp eax, STATUS_SUCCESS
```

```
    jnz .simlinkfail
    mov rax, [pDriverObject]
    virtual at rax
   .rax DRIVER_OBJECT
    end virtual

    mov rdx, DispatchCreate
    mov [.rax.MajorFunction+IRP_MJ_CREATE*8], rdx
    mov rdx, DispatchClose
    mov [.rax.MajorFunction+IRP_MJ_CLOSE*8], rdx
    mov rdx, DispatchControl
    mov [.rax.MajorFunction+IRP_MJ_DEVICE_CONTROL*8], rdx
    mov rdx,  DriverUnloadHandler
    mov [.rax.DriverUnload], rdx

    mov [status], STATUS_SUCCESS
    jmp .END
   .simlinkfail:
    invoke IoDeleteDevice, [DeviceObject]
   .END:

    mov eax, [status]
    ret
endp
```

Код драйвера по сравнению с 32-битной версией тоже не очень сильно изменился.

При программировании в нулевом кольце требование о выравнивании стека остаётся в силе. Функция `DispatchControl` тому подтверждение: при сохранении в стеке регистров RBX, RSI, RBI стек становится не выровненным, и для его выравнивания необходимо либо уменьшить регистр RSP на 8 байт, либо поместить в стек ещё что-нибудь. Последнее было бы неправильным шагом, т. к. сохранение в стеке чего-либо – это обращение к памяти, а оно всегда занимает больше времени, чем просто уменьшение регистра на восемь. Возможно, такое утверждение было бы спорным в отношении современных процессоров, но тем не менее всегда стоит помнить, что трансляция адреса в 64-битном режиме у процессора занимает вдвое больше времени, чем в 32-битном. Даже с учетом кэширования данных и TLB (кэш адресов, к которым недавно обращалась программа), если можно избежать обращения к памяти, то обязательно следует воспользоваться этой возможностью. Правило минимизации обращений к памяти является основным при оптимизации кода 64-разрядных программ.

При использовании локальных переменных нет необходимости выравнивать стек, т. к. макрос proc при выделении места под локальные переменные округляет размер всех локальных данных до числа кратного 16.

Код приложения, управляющего драйвером, почти не изменился за исключением того, что оно теперь получает содержимое дескриптора шлюза прерывания, отвечающего на страничное нарушение (вектор 13).

Полный исходный код примеров из данного раздела можно найти на компакт-диске, прилагающемся к книге, в папке part5.

5.2.5. Отладка приложений в Win64

В отличие от 32-разрядных систем, 64-разрядные системы Windows не имеют широкого спектра отладочных программ – как для третьего кольца, так и для нулевого. Наиболее приемлемым для 64-разрядных систем Windows является отладчик WinDBG, опять же как для третьего, так и для нулевого кольца (при отладке ядра по-прежнему требуются два компьютера или использование виртуальной машины).

5.2.6. Резюме

В данном разделе речь шла об основах программирования в 64-битных версиях Windows. Раздел полностью посвящен практической составляющей; нами были написаны оконная программа для третьего кольца и драйвер для нулевого кольца.

6 МНОГОПРОЦЕССОРНЫЕ СИСТЕМЫ

6.1. Работа с APIC

Итак, мы изучили два основных режима процессора, научились переключать режимы процессора. Но прогресс не стоит на месте – сейчас почти каждый компьютер оснащён процессором с несколькими ядрами, а серверы и вовсе оснащены несколькими процессорами. В настоящей главе речь пойдёт о межпроцессорном взаимодействии.

Основой межпроцессорного взаимодействия является *расширенный программируемый контроллер прерываний*, или *APIC* (Advanced Programmable Interrupt Controller). Поэтому перед изучением межпроцессорного взаимодействия необходимо изучить работу с APIC.

6.1.1. Общий обзор

Контроллер APIC является заменой устаревшему контроллеру прерываний, составленному из двух микросхем i8259A. По правде говоря, название APIC не вполне корректно, т. к. APIC использовался на процессорах Pentium 1, 2, 3 (P6 Family), а версия APIC, которая используется на процессорах Pentium 4 и Xeon и о которой будет идти речь ниже, является более новой и называется xAPIC. xAPIC по сравнению с APIC имеет ряд новшеств и расширений (отсюда и первая буква «x» в названии). Далее под термином «APIC» будет подразумеваться xAPIC.

По-умолчанию APIC работает в режиме совместимости со старым контроллером прерываний i8259A. APIC состоит из двух частей: локального контроллера и контроллера ввода/вывода I/O APIC. *Локальный APIC* (Local APIC) есть у каждого процессора (ядра). Локальный APIC содержит основной функционал расширенного программируемого контроллера прерываний. *I/O APIC* – один на все процессоры; он принимает прерывания от внешних устройств и посылает сигналы локальным APIC для дальнейшей обработки. Расширенный программируемый контроллер прерываний является основой межпроцессорного взаимодействия, т. к. без него была бы невозможна либо очень сильно затруднена обработка прерываний в многопроцессорной системе. А значит, без APIC было бы невозможным взаимодействие процессоров в многопроцессорной системе.

Как же протекает работа с APIC? Эта работа ведётся через регистры, отображённые на память. Что значит *регистр, отображённый на память*? Это значит, что, читая (или записывая) из определённого адреса памяти, мы читаем (или записываем) из какого-либо регистра некоторого устройства. Разумеется, адреса, на

которые отображены регистры APIC (или других устройств), находятся в специальной области памяти, обычно – в старшем гигабайте памяти, специально отведённом для этих целей разработчиками компании Intel. Все регистры APIC отображены на память, как правило на адреса после 0FEE00000h, но этот адрес можно изменить через специальный MSR-регистр.

Как уже было сказано, основной функционал находится в локальном APIC. Последний позволяет:

- обрабатывать прерывания из внутренних источников (таймер, температурный датчик);
- обрабатывать прерывания из внешних источников (от внешних устройств через I/O APIC);
- посылать межпроцессорные прерывания (подробнее об этой функции будет рассказано в разделе 6.2);
- вести обработку внутренних ошибок.

Ниже будет рассказано про обработку прерываний с помощью расширенного программируемого контроллера прерываний, а в качестве примера будет написан пример программы, принимающей прерывания от таймера локального APIC, а также от клавиатуры через I/O APIC.

6.1.2. Включение APIC

Узнать, существует ли локальный APIC (либо поддерживается ли он процессором), можно с помощью команды `CPUID`; перед вызовом команды в EAX должно находиться число 1. Если после вызова команды `CPUID` в девятом бите регистра EDX находится единица, то локальный APIC существует. Поскольку сейчас почти все компьютеры оборудованы расширенным контроллером прерываний, проверка наличия APIC становится исключительной формальностью – ею можно даже пренебречь.

За включение APIC отвечает MSR-регистр `IA32_APIC_BASE` и, как видно из названия, с его помощью можно получить/изменить местоположение регистров контроллера в памяти. Регистр `IA32_APIC_BASE` имеет индекс 1Bh. На рис. 6.1 приведена его схема.

Поле **APIC Base** содержит базовый адрес регистров APIC, базовый адрес всегда выровнен на границу страницы, поэтому значение младших 12 бит не имеет значения. После включения или перезагрузки процессора базовый адрес равен 0FEE00000h. Максимальная разрядность адреса зависит от модели процессора, но не может быть более 52.

Поле **APIC global enable/disable** отвечает за глобальное включение/выключение контроллеров APIC (локального APIC и IO APIC). Единица в этом бите включает контроллер, ноль – выключает контроллер.

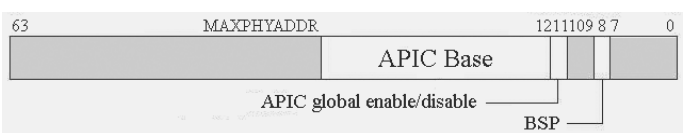


Рис. 6.1. Регистр IA32_APIC_BASE

Поле **BSP** сообщает нам, является ли текущий процессор BSP, т. е. запускается ли он первым после включения компьютера или перезагрузки. Подробнее об этом флаге будет рассказано в разделе 6.2.

Следует отметить, что работать с APIC необходимо после его включения, т. к. работа с APIC в выключенном состоянии бессмысленна. Более того, выключение APIC сбрасывает его в первоначальное состояние.

Изменять базовый адрес APIC следует именно во время его включения – путём занесения нужного адреса в поле APIC Base.

6.1.3. Local APIC ID

Каждый локальный APIC имеет свой ID, по которому его можно идентифицировать; этот ID, по сути, является его именем. Local APIC ID используется I/O APIC при распределении внешних прерываний между процессорами. ID локального APIC задаётся регистром Local APIC ID, который находится по адресу +20h от стартового адреса Local APIC (по умолчанию 0FEE00020h). Структура этого регистра приведена на рис. 6.2.

Рис. 6.2. Регистр Local APIC ID

Как видно из рисунка, под ID выделено 8 бит, а следовательно, на материнской плате может быть установлено до 256 процессоров (или ядер). Следует заметить, что изменение Local APIC ID не всегда поддерживается процессором.

6.1.4. Локальная векторная таблица

У каждого локального контроллера есть своя собственная векторная таблица. В *локальной векторной таблице* (LVT, local vector table) задаются векторы для каждого прерывания и тип доставки. Таблица состоит из шести регистров, все они 32-битные. Перечислим эти регистры (в скобках указаны смещения от базового адреса Local APIC):

- LVT Timer Register (+320h) – содержит информацию для обработки прерываний APIC Timer;
- LVT Thermal Monitor Register (+330h) – содержит информацию для обработки прерываний температурного датчика;
- LVT Performance Counter Register (+340h) – содержит информацию для обработки прерываний датчика производительности;
- LVT LINT0 Register (+350h) и LVT LINT1 Register (+360h) – содержат информацию для обработки прерываний из входов LINT0 и LINT1;
- LVT Error Register (+370h) – содержит информацию для обработки прерывания, возникающего в случае, когда контроллер обнаруживает внутреннюю ошибку.

Источники прерываний LINT0 и LINT1 могут быть присоединены к внешним источникам, а именно через I/O APIC к внешним устройствам. Общая схема локальной векторной таблицы приведена на рис. 6.3.

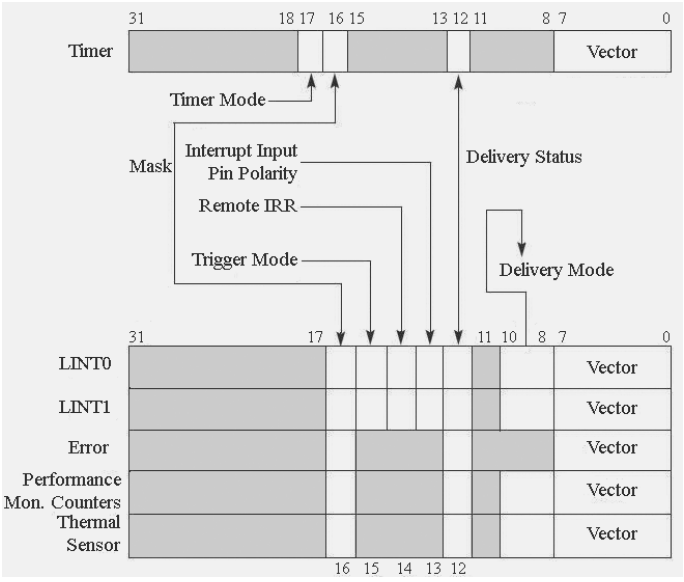


Рис. 6.3. Локальная векторная таблица

Поле **Vector (**биты 0–7) содержит вектор прерывания (номер обработчика в IDT), который будет вызван при срабатывании данного источника прерывания. Значение вектора меньше 16 считается недействительным и приводит к ошибке. Следует отметить, что значения векторов 16–31, которые тоже зарезервированы Intel, локальный APIC не считает недействительными.

Поле **Delivery Mode** (биты 8–10) содержит тип доставки сигнала прерывания.

- Fixed (000) – вызывается обработчик прерывания, находящийся в указанном векторе;
- SMI (010) – прерывание доставляется как специальное прерывание SMI. После его получения процессор переходит в *режим системного управления* (System Management Mode, SMM). Вектор должен быть равен нулю (для совместимости с будущими программами);
- NMI (100) – прерывание доставляется как немаскируемое (Non Masked Interrupt). Если процессор находится в защищённом или 64-битном режиме, то будет вызван второй вектор из IDT. Номер вектора игнорируется;
- INIT (101) – отправка сигнала INIT процессору. Подробнее об этом типе прерывания будет рассказано позже. Вектор должен быть равен нулю (для совместимости с будущими программами);
- ExtINT (111) – обычно используется для LINT0 и LINT1: это значит, что прерывание привязывается к внешнему источнику (к контроллеру, совместимому с 8259A).

Поле **Delivery Status** (бит 12) предназначено только для чтения и указывает текущий статус доставки прерывания. Ноль означает, что сейчас ничего не происходит

либо прерывание вызвано и успешно принято процессором на обработку. Единица означает, что прерывание доставлено процессору, но процессор ещё не принял его на обработку.

Поле **Trigger Mode** (бит 15) актуально только для типа доставки Fixed и только для LINT0 и LINT1. С помощью этого бита можно установить чувствительность линии либо к фронту сигнала (edge sensitive) либо к уровню сигнала (level sensitive). Когда чувствительность установлена к фронту сигнала, запрос на линии фиксируется в момент, когда сигнал на этой линии переходит из 0 в 1 (или из 1 в 0, в зависимости от поля Interrupt Pin Polarity), и остаётся зафиксированным до тех пор, пока не будет обработан, даже если после фиксации линия вновь перешла из 1 в 0. Если линия чувствительна к уровню сигнала (level sensitive), запрос от неё активен только в то время, пока активна сама линия, т. е. когда на ней находится 1 (или 0, в зависимости от поля Interrupt Pin Polarity). Если линия «падает» в ноль до того, как началась обработка её запроса, считается, что запроса не было вообще.

Поле **Interrupt Pin Polarity** (бит 13) устанавливает инверсивность линии. Если в этом поле ноль, то линия не инверсная, т. е. запрос присутствует, когда на линии единица. Если в этом поле единица, то линия считается инверсной, т. е. запрос присутствует, когда на линии ноль.

Поле **Remote IRR** (бит 14, только для чтения) актуально только для прерываний, чувствительных к уровню сигнала. Этот бит выставляется, когда процессор принимает прерывание на обработку, и обнуляется, когда обработчик прерывания производит операцию EOI.

Поле **Mask** (бит 16) отвечает за маскировку прерывания. Если в этом бите содержится ноль, то прерывание не замаскировано, а если единица, то замаскировано. По умолчанию все прерывания в локальной векторной таблице замаскированы.

Поле **Timer Mode** (бит 17 в LVT Timer Register) устанавливает режим работы таймера. Если оно равно нулю, то режим работы однократный; если единица, то многократный.

Особого внимания заслуживают источники прерываний LINT0 и LINT1. Это, по сути, определяемые операционной системой источники прерываний. Например, вход LINT0 можно настроить на получение сигналов SMI, а вход LINT1 – на получение прерываний от внешних устройств с контроллера, совместимого с 8259A. А поскольку эти настройки будут индивидуальными для каждого процессора, можно произвести их таким образом, чтобы каждый процессор отвечал за свой тип прерываний. Распределение типов прерываний между процессорами – это ещё одно преимущество APIC.

По-умолчанию все шесть источников прерываний запрещены, бит Mask у всех установлен в единицу.

6.1.5. Local APIC Timer

Каждый локальный контроллер может обрабатывать прерывания от системного таймера. Прерывания от системного таймера выделены как отдельный источник прерывания в локальной векторной таблице.

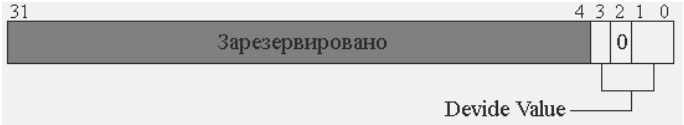


Рис. 6.4. Divide Configuration Register (+3E0h)

За настройку этого источника прерывания отвечают регистры (в скобках указаны смещения от базового адреса Local APIC):

- LVT Timer Register (+320h);
- Divide Configuration Register (+3E0h);
- Initial Count Register (+380h);
- Current Count Register (+390h).

Настройка самого источника прерывания производится через регистр LVT Timer Register. В нём мы указываем вектор прерывания, режим работы таймера и маскировку прерывания. Режимов работы таймера два: однократный и многократный; если в семнадцатом бите находится ноль, то режим работы однократный, а если единица, то многократный. За разрешение этого типа прерывания отвечает бит маскировки (бит 16): если ноль – разрешено, если единица – запрещено.

С помощью Divide Configuration Register можно установить делитель частоты системного таймера. Схема этого регистра приведена на рис. 6.4.

В зависимости от значений в поле **Divide Value** (биты 0, 1 и 3) в этом регистре частота генерации прерываний будет делиться на следующие значения:

- 000 – деление на 2;
- 001 – деление на 4;
- 010 – деление на 8;
- 011 – деление на 16;
- 100 – деление на 32;
- 101 – деление на 64;
- 110 – деление на 128;
- 111 – нет деления.

С помощью Initial Count Register можно задать начальное значение счётчика. А текущее значение счётчика можно прочитать с использованием Current Count Register. После того как в Initial Count Register будет занесено какое-либо значение, таймер считается включённым, т. е. момент занесения значения в Initial Count Register считается включением таймера.

В однократном режиме, когда таймер включается, в регистр Current Count Register заносится значение из Initial Count Register и уменьшается на единицу при каждом срабатывании таймера – в зависимости от делителя. Когда значение Current Count Register становится равным нулю, то происходит вызов прерывания по указанному вектору.

При многократном режиме, когда значение в Current Count Register обнуляется, после вызова прерывания в Current Count Register снова заносится значение из Initial Count Register, и цикл повторяется.

6.1.6. Обработка прерываний

Обработка прерываний Local APIC происходит по следующему алгоритму:

1. Если Local APIC получает сигнал о прерываниях NMI, SMI, INIT, ExtINT и Start-up IPI (подробнее об этом прерывании мы поговорим в следующем разделе), то незамедлительно посылает процессору прерывание на обработку.
2. Если Local APIC получает прерывание Fixed, то на основе приоритетов, используя регистры IRR, ISR, TPR и PPR, принимает решение о посылке процессору прерывания на обработку.
3. О том, что прерывание обработано, Local APIC извещает операция EOI (запись в регистр EOI). После операции EOI Local APIC отправляет на обработку следующее прерывание, если таковое стоит в очереди. (Для прерываний NMI, SMI, INIT, ExtINT и Start-up IPI операция EOI не требуется.)

Как уже было сказано, обработка прерываний происходит на основе приоритетов. Приоритет прерывания получается делением вектора прерывания на 16 и округлением до ближайшего целого числа. Поскольку действительными считаются вектора от 32 до 255, то приоритеты прерываний варьируются в диапазоне от 2 до 15. Таким образом, в каждый класс приоритета попадают 16 векторов прерываний.

Чтобы указать Local APIC, какие прерывания принимать, нужно использовать регистр TPR (Task Priority Register) – он имеет смещение 80h от базового адреса Local APIC (по умолчанию 0FEE00080h); его схема приведена на рис. 6.5.

Поле **Task Prioriry**, биты 4–7 регистра TPR, содержит уровень приоритета прерываний, принимаемых Local APIC. Если уровень приоритета больше числа, указанного в этом поле, то прерывание принимается на обработку. Также можно указать минимальный подкласс прерывания (остаток от деления на 16). С помощью этого регистра операционная система может временно блокировать прерывания низкого приоритета.

В long mode есть дополнительный способ доступа к регистру TPR – через регистр CR8. Приоритет задаётся первыми четырьмя битами, остальные зарезервированы. При записи в регистр CR8 происходит следующее: `APIC.TPR[7:4] = CR8[3:0], APIC.TPR[3:0] = 0`. При чтении из регистра CR8 мы получаем значение `APIC.TPR[7:4]`.

Также при обработке прерываний Local APIC использует регистр PPR (Processor Priority Register). Этот регистр доступен только для чтения и расположен по смещению 0A0h от базового адреса Local APIC (по умолчанию 0FEE000A0h). Формат регистра PPR аналогичен TPR, его структура приведена на рис. 6.6.

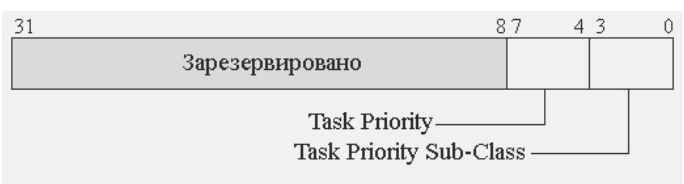


Рис. 6.5. - Регистр TPR (+80h)

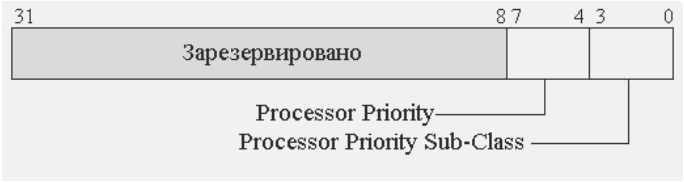


Рис. 6.6. Регистр PPR (+0A0h)

Значение этого регистра вычисляется автоматически по следующему алгоритму.

```
IF TPR[7:4] ≥ ISRV[7:4]
    THEN
            PPR[7:0] ← TPR[7:0]
    ELSE
            PPR[7:4] ← ISRV[7:4]
            PPR[3:0] ← 0
```

Значение ISRV равно максимальному номеру вектора прерывания, которое в настоящее время обрабатывается процессором. Максимальный номер вектора прерывания равен порядковому номеру самого старшего бита равного единице в регистре ISR (подробнее о нём будет рассказано позже).

При обработке прерываний Local APIC использует следующие доступные только для чтения регистры: Interrupt Request Register (IRR), In-Service Register (ISR). Оба регистра являются 256-битными и состоят из восьми 32-битных. Регистры IRR расположены по смещениям 200h, 210h, 220h...270h от базового адреса Local APIC. Регистры ISR расположены по смещениям 100h, 110h, 120h...170h от базового адреса Local APIC.

Как же происходит обработка прерывания? Когда Local APIC принимает сигнал о прерывании, то принимает его либо отбрасывает, руководствуясь значением в регистре PPR. Если Local APIC принимает сигнал о прерывании, то он выставляет соответствующий бит в регистре IRR (бит под номером, соответствующим вектору прерывания). Как только процессор будет готов обработать прерывание (а именно флаг IF в регистре флагов процессора станет равным единице), Local APIC очищает самый старший бит в регистре IRR (запросов на прерывание в регистре IRR может накопиться несколько) и устанавливает этот же самый бит в регистре ISR. Регистр ISR говорит о том, какой вектор прерывания в данный момент обрабатывается процессором. После обработки прерывания обработчик должен произвести операцию end-of-interrupt (EOI).

Операция EOI осуществляется путём записи нулевого значения в регистр EOI. Регистр EOI находится по смещению 0B0h от базового адреса Local APIC (по умолчанию 0FEE000B0h). Запись в этот регистр любого значения приводит к операции EOI, но для возможной совместимости с будущими программами необходимо записывать ноль в этот регистр. (Запись ненулевого значения приведёт к исключению общей защиты.)

После осуществления операции EOI Local APIC очищает самый старший бит со значением 1, а затем находит самый старший выставленный бит в регистре IRR, очищает его и выставляет в регистре ISR. Далее процесс повторяется.

Как только Local APIC получает сигнал о прерывании приоритетом выше, чем то, которое обрабатывается процессором, причем прерывания разрешены на процессоре (т. е. флаг IF в регистре флагов выставлен), то прерывание отправляется процессору на обработку. Таким образом, в регистре ISR может быть выставлено более одного бита.

6.1.7. Работа с I/O APIC

Итак, мы изучили работу с Local APIC. Local APIC является посредником между процессором, системной шиной и I/O APIC. I/O APIC в свою очередь является посредником между внешними устройствами и процессорами. I/O APIC в отличие от контроллера 8259A поддерживает 24 линии аппаратных прерываний, в то время как контроллер 8259A поддерживал только 16 (более новые версии контроллера APIC могут поддерживать более 24 линий).

Назначение линий аппаратных прерываний I/O APIC зависит от модели материнской платы; точно известны назначения следующих линий:

1. INTIN1 – клавиатура PS/2;
2. INTIN12 – мышь PS/2;
3. INTIN16-19 – устройства PCI;
4. INTIN23 – линия прерывания SMI.

Возможно, что линии INTIN16-19 прерываний могут быть переназначены на некоторых моделях материнских плат.

Работа с I/O APIC тоже осуществляется через спроецированные на память регистры, но теперь регистров всего два. Регистры I/O APIC обычно спроецированы на адреса начиная с 0FEC00000h. Первый регистр называется IOREGSEL – он нужен для выбора регистра, с которым мы собираемся работать. Регистр IOREGSEL по умолчанию расположен по адресу 0FEC00000h. Второй регистр называется IOWIN – это окно, через которое производится занесение и получение данных из выбранного регистра; по умолчанию IOWIN расположен по адресу 0FEC00010h (+10h относительно базы I/O APIC).

Регистр IOAPICID, индекс 0. Через этот регистр можно задать ID контроллера. ID находится в битах 24–27, остальные биты зарезервированы.

Регистр IOAPICVER, индекс 1, предназначен только для чтения; содержит версию I/O APIC и максимальное количество элементов в таблице перенаправлений (количество линий прерываний). Биты 0–7 содержат номер версии. Биты 16–23 содержат максимальное число элементов в таблице перенаправлений (количество элементов минус 1).

Регистры IOREDTBL0-23 являются элементами таблицы перенаправлений. В таблице перенаправлений прерываний содержится информация (вектор, тип доставки и т. д.) об обработке прерываний из соответствующей линии прерывания. Каждый элемент таблицы представляет собой 64-битный регистр. Индексы регистров указаны в табл. 6.1.

Таблица 6.1. Индексы регистров таблицы перенаправлений

Регистр	Индекс старшей части	Индекс младшей части
IOREDTBL0	10h	11h
IOREDTBL1	12	13h
IOREDTBL2	14	15h
IOREDTBL3	16	17h
IOREDTBL4	18	19h
IOREDTBL5	1A	1Bh
IOREDTBL6	1C	1Dh
IOREDTBL7	1E	1Fh
IOREDTBL8	20	21h
IOREDTBL9	22	23h
IOREDTBL10	24	25h
IOREDTBL11	26	27h
IOREDTBL12	28	29h
IOREDTBL13	2A	2Bh
IOREDTBL14	2C	2Dh
IOREDTBL15	2E	2Fh
IOREDTBL16	30	31h
IOREDTBL17	32	33h
IOREDTBL18	34	35h
IOREDTBL19	36	37h
IOREDTBL20	38	39h
IOREDTBL21	3A	3Bh
IOREDTBL22	3C	3Dh
IOREDTBL23	3E	3Fh

Все регистры в таблице перенаправлений имеют одинаковую структуру. Структура регистра в таблице перенаправлений изображена на рис. 6.7.

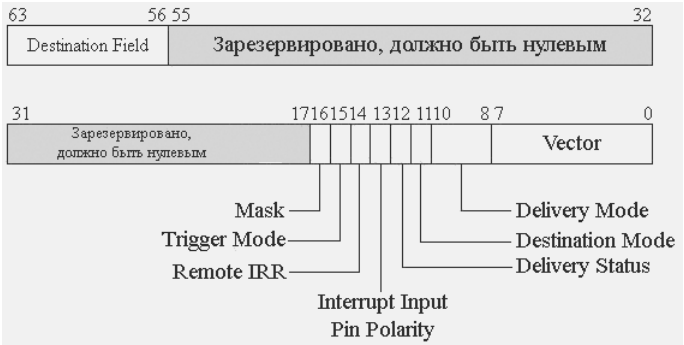
Поле **Destination Field**, биты 56–63, зависит от поля Destination Mode. Если в поле Destination Mode находится 0, то в этом поле должен содержаться целевой Local APIC ID. Если в поле Destination Mode находится 1, то в поле должна содержаться маска для выбора процессоров (подробнее об этом пойдет речь в следующем разделе).

Поле **Mask**, бит 16, отвечает за маскировку прерывания; если поле равно единице, то прерывание замаскировано.

Поле **Trigger Mode**, бит 15, задаёт режим чувствительности линии: если 0, то чувствительность к фронту, если 1, то чувствительность к уровню.

Поле **Remote IRR**, бит 14, предназначено только для чтения и актуально только для прерываний, чувствительных к уровню. Если поле равно 1, то Local APIC принял прерывание на обработку. Поле сбрасывается, когда Local APIC производит операцию EOI.

Рис. 6.7. Формат регистра таблицы перенаправлений I/O APIC



Поле **Interrupt Input Pin Polarity**, бит 13, задаёт инверсивность линии. Если выставлена 1, то линия инверсивная.

Поле **Delivery Status**, бит 12, – только для чтения. Для этого прерывания механизм посылки был инициирован, но посылка временно задержана APIC по причине занятости шины, или же принимающий Local APIC не может принять прерывание в данное время.

Поле **Destination Mode**, бит 11, устанавливает тип адресата прерывания. Если в этом поле 0, то включён Physical Mode; в поле Destination Field должен содержаться APIC ID целевого контроллера. Если в этом поле 1, то включен Logical Mode; в поле Destination Field должна содержаться маска для выбора процессоров (подробнее об этом будет рассказываться в следующем разделе).

Поле **Delivery Mode**, биты 8–10, задаёт тип доставки прерывания:

- Fixed (000). Доставляет этот сигнал до INTR-вывода всех процессоров, выбранных в качестве адреса получателя. Чувствительность может быть как к фронту, так и к уровню сигнала.
- Lowest Priority (001). Доставляет этот сигнал до INTR-вывода всех процессоров, которые находятся в режиме низкого приоритета и указаны в списке адреса получателя. Чувствительность может быть как к фронту, так и к уровню сигнала.
- SMI (010). Прерывание доставляется процессорам как SMI-прерывание. Чувствительность должна быть к фронту сигнала, а вектор должен быть равен нулю для совместимости с будущими программами.
- NMI (100). Прерывание доставляется процессорам как NMI-прерывание. Вектор прерывания игнорируется, а чувствительность должна быть к фронту сигнала.
- INIT (101). Процессорам отправляется сигнал INIT. После доставки этого прерывания все Local APIC, к которым произошло обращение, будут находиться в состоянии INIT. Вектор прерывания игнорируется, а чувствительность должна быть к фронту сигнала.
- ExtINT (111). Доставляет этот сигнал прерывания до выводов INTR всех выбранных процессоров. Прерывание распознается как возникшее от внешнего

контроллера прерывания, совместимого с 8259A. Ответный сигнал INTA согласуется с доставкой ExtINT и перенаправляется к внешнему контроллеру, который выдает вектор прерывания. Чувствительность должна быть к фронту сигнала. При данном типе доставки I/O APIC становится посредником между Local APIC и контроллером 8259A.

- Значения (011) и (110) зарезервированы.

Поле **Vector**, биты 0–7, содержит номер вектора прерывания. Действительными считаются значения в диапазоне 10h–0FEh.

6.1.8. Практика

В практический части данного раздела мы напишем программу, которая переходит в защищённый режим, после чего инициализирует контроллеры APIC для получения прерываний таймера Local APIC и прерываний клавиатуры. Программа будет написана на основе каркаса, созданного нами в разделе 2.1.

Ну что ж, приступим. Для упрощения и повышения «читабельности» кода адреса и индексы регистров APIC целесообразно объявить как константы и вынести в отдельный файл. Вначале нам надо вывести сообщение о том, что мы находимся в защищённом режиме, включить APIC путём установления в единицу одиннадцатого бита регистра IA32_APIC_BASE. А после вывести сообщение о том, что APIC включён (листинг 6.1).

Листинг 6.1. Инициализация расширенного контроллера прерываний

```
include 'pmstructures.asm';
include 'APIC_defs.asm'

IDT_size equ IDT_END-IDT

  START_CODE:

   mov esi, message1
   mov al, 0
   mov ah, 0
   mov bl, "5"
   call OutText

   lidt fword [IDTR]

; init APIC
     mov ecx, IA32_APIC_BASE
     rdmsr
     bts eax, 11
     wrmsr

;APIC enabled
   mov esi, message2
   mov al, 0
```

```
mov ah, 1
mov bl, "5"
call OutText
```

Итак, APIC включён. Теперь нам надо настроить Local APIC Timer. Для этого надо настроить регистры LVT Timer Register, Divide Configuration Register, Initial Count Register. Настройка таймера Local APIC приведена в листинге 6.2.

Листинг 6.2. Инициализация таймера Local APIC

```
; enabling LAPIC timer
   mov dword [APIC_LVT_Timer_REG_DEF], ((1 shl 17) or 20h)
   mov dword [APIC_Timer_Divide_Configuration_REG_DEF], 1001b
   mov dword [APIC_Timer_Initial_Count_REG_DEF], 2083333

   mov esi, message3
   mov al, 0
   mov ah, 2
   mov bl, "5"
   call OutText
```

Таймер настроен на вызов 32 (20h) вектора прерывания и на многократный режим работы. Частота таймера нигде не документирована – можно выдвинуть предположение, что она зависит от тактовой частоты процессора. Если в результате работы скорость счётчика на экране будет слишком большая, её можно будет изменить путём изменения делителя и значения в регистре Initial Count Register (сейчас значение делителя равно 64, а значение Initial Count – 2083333). Следующим шагом станет настройка I/O APIC (листинг 6.3).

Листинг 6.3. Настройка I/O APIC

```
mov dword [IOAPIC_IOREGSEL_REG_DEF], IOAPIC_IOAPICVER
mov eax, dword [IOAPIC_IOWIN_REG_DEF]
shr eax, 16
inc eax     ;
mov ebx, 10
mov esi, Max_RTE_Str
call dword_to_STR

mov esi, message4
mov al, 0
mov ah, 3
mov bl, "5"
call OutText
```

В вышеприведённом коде мы получаем значение регистра IOAPICVER. Нам оно нужно для получения количества элементов в таблице перенаправлений контроллера. В конце мы выводим сообщение о количестве элементов в таблице. Функция `dword_to_STR` производит преобразование числа в строку. Теперь нам осталось только настроить линии аппаратных прерываний от клавиатуры (листинг 6.4).

Листинг 6.4. Настройка I/O APIC на приём прерываний от клавиатуры

```
mov eax, 1
cpuid
and ebx, 0FF000000h ; high byte = Local APIC ID

mov dword [IOAPIC_IOREGSEL_REG_DEF], IOAPIC_IOAPICID
mov dword [IOAPIC_IOWIN_REG_DEF], 2000000h

mov dword [IOAPIC_IOREGSEL_REG_DEF], IOAPIC_IOREDTBL1_hi
mov dword [IOAPIC_IOWIN_REG_DEF], ebx

mov dword [IOAPIC_IOREGSEL_REG_DEF], IOAPIC_IOREDTBL1_low
mov dword [IOAPIC_IOWIN_REG_DEF], 21h

mov esi, message5
mov al, 0
mov ah, 4
mov bl, "5"
call OutText

sti

jmp $
```

Вначале мы получаем Local APIC ID нашего процессора с помощью инструкции `CPUID`. При вызове `CPUID` с единицей в регистре EAX в старший байт регистра EBX будет сохранён Local APIC ID текущего процессора. После получения Local APIC ID устанавливается IO APIC ID. Затем настраивается элемент в таблице перенаправлений, отвечающий за прерывания клавиатуры. В нём нам надо установить ID локального APIC, которому будут доставляться прерывания и вектор прерывания. Все остальные поля можно обнулить. В конце выводится сообщение об инициализации I/O APIC, разрешаются прерывания и запускается бесконечный цикл. Теперь надо написать обработчики прерывания таймера и клавиатуры.

Листинг 6.5. Обработчик прерываний таймера

```
counter dd 0

LVT_APIC_Timer_handler:

   push esi
   push eax
   push ebx
   inc dword [counter]
   mov eax, [counter]
   mov ebx, 10
   mov esi, TimerCounterStr
   call dword_to_STR
```

```
   mov esi, TimerCounterStr
   mov al, 0
   mov ah, 5
   mov bl, "5"
   call OutText

   mov dword [APIC_EOI_REG_DEF], 0 ;EOI
   pop ebx
   pop eax
   pop esi

   iretd
```

В обработчике таймера производится увеличение счётчика на единицу и вывод его на экран (с предварительным преобразованием в строку). В конце обработчик производит операцию EOI.

Листинг 6.6. Обработчик прерываний клавиатуры

```
kbrd_counter dd 0

Keyboard_handler:

   push esi
   push eax
   push ebx
   inc dword [kbrd_counter]

   mov eax, [kbrd_counter]
   mov ebx, 10
   mov esi, KBRDCounterStr
   call dword_to_STR

   mov esi, KBRDCounterStr
   mov al, 0
   mov ah, 6
   mov bl, "5"
   call OutText

   in al, 060h
   dec  al
   mov  ah, al
   and  ah, 80h
   jz @f
   in al, 061h
   or al, 80
   out  061h, al
   xor  al, 80
   out  061h, al
```

```
@@:
 mov dword [APIC_EOI_REG_DEF], 0 ;EOI
 pop ebx
 pop eax
 pop esi

 iretd
```

В обработчике клавиатуры производятся действия, аналогичные тем, которые имеют место в обработчике таймера, за исключением одного момента. Когда обрабатывается «отпускание» кнопки клавиатуры (об этом свидетельствует единица в старшем бите в порте 60h), производится информирование клавиатуры о том, что нажатие обработано. Осуществляется это путём установки старшего бита в единицу в порте 61h на короткое время.

Код функций вывода сообщения на экран и преобразования числа в строку, а также код объявления таблицы прерываний приводить здесь не имеет смыла, ибо они тривиальны. Полный исходный код программы можно найти на диске, прилагающемся к этой книге (папка part6).

6.1.9. Резюме

В этом разделе мы изучили принципы работы с расширенным программируемым контроллером прерываний. Нами были рассмотрены основы работы с локальным APIC и I/O APIC. Итак, теперь мы приобрели все базовые знания для изучения межпроцессорного взаимодействия, о котором и пойдёт речь в следующем разделе.

6.2. Межпроцессорное взаимодействие

В разделе 6.1 нами были изучены принципы работы с APIC. Благодаря возможности посылать межпроцессорные прерывания APIC является основой межпроцессорного взаимодействия.

В данном разделе будут описаны механизм посылки межпроцессорных прерываний и алгоритм инициализации межпроцессорной системы. В практической части мы напишем программу, производящую инициализацию всех доступных процессоров в многопроцессорной компьютерной системе.

6.2.1. Общий обзор

Перед тем как приступить к рассмотрению механизма межпроцессорного взаимодействия, необходимо разобраться, какие же бывают типы многопроцессорных систем. Есть три основных типа многопроцессорных систем:

1. Многопроцессорная система, базирующаяся на материнской плате с несколькими сокетами для процессоров.
2. Процессор с несколькими ядрами.
3. Процессор с несколькими виртуальными ядрами (технология Hyper-Threading).

Теперь подробнее о каждой. Первый тип – это самый простой тип многопроцессорной системы: у нас имеется материнская плата с несколькими сокетами под

процессоры. Этот тип очень часто применяется при создании серверных компьютерных систем (интернет-серверов, серверов баз данных и др.)

Второй тип – наиболее распространённый среди пользовательских персональных компьютеров. Этот тип многопроцессорной системы представляет собой материнскую плату с сокетом под один процессор; притом процессор может быть с двумя или более ядрами. *Многоядерный процессор* – это не что иное, как несколько процессоров в одном кристалле. Такой способ является менее затратным, т. к. нет необходимости в дорогой материнской плате с несколькими сокетами. У многопроцессорной системы данного типа есть только один недостаток, связанный с тем, что ядра делят между собой одну системную шину, а это уменьшает скорость доступа каждого ядра к памяти и внешним устройствам.

Третий вид многопроцессорных систем представляет собой один процессор, поддерживающий *технологию HyperThreading* с несколькими (как правило, двумя) виртуальными ядрами. Физически это один процессор с одним ядром, но для программного обеспечения создаётся иллюзия, что ядер несколько. Использование нескольких виртуальных ядер даёт прирост производительности в некоторых типах решаемых задач почти при той же стоимости компьютерной системы. Такой тип компьютерной системы использовался, когда процессоры с полноценными исполнительными ядрами были очень дорогостоящими; сейчас такие процессоры используются всё реже.

Все три типа могут комбинироваться друг с другом. К примеру, есть материнская плата с двумя сокетами под процессоры. В одном сокете стоит процессор с поддержкой технологии HyperThreading с двумя виртуальными ядрами, а в другом сокете – процессор с четырьмя полноценными исполнительными ядрами; в сумме на этой процессорной системе будет шесть процессоров. Именно процессоров, а не ядер! В программном смысле ядро и HyperThreading-ядро – это полноценный процессор со своим собственным набором регистров (MSR-регистры и системные регистры управления у каждого тоже свои) и устройств (к примеру, Local APIC у каждого процессора свой, даже если он является HyperThreading-ядром). Общее у всех процессоров только одно – системная память, устройства и контроллеры, находящиеся на материнской плате.

У всех трёх вышеприведённых типов многопроцессорных систем, как ни странно, одинаковый протокол инициализации, что, впрочем, очень удобно, т. к. программе нет необходимости выяснять, на каком же типе многопроцессорной системы работает программа. Программа, инициализирующая процессоры, будет одинаково работать как на процессоре с HyperThreading, так и на многопроцессорной системе с несколькими физическими процессорами.

Подытожив всё вышесказанное, можно сделать следующий вывод: для программы или операционной системы нет никакой разницы, на каком процессоре она работает.

6.2.2. Межпроцессорные прерывания

Для осуществления межпроцессорных прерываний разработчиками компании Intel был создан специальный регистр – Interrupt Command Register (далее ICR).

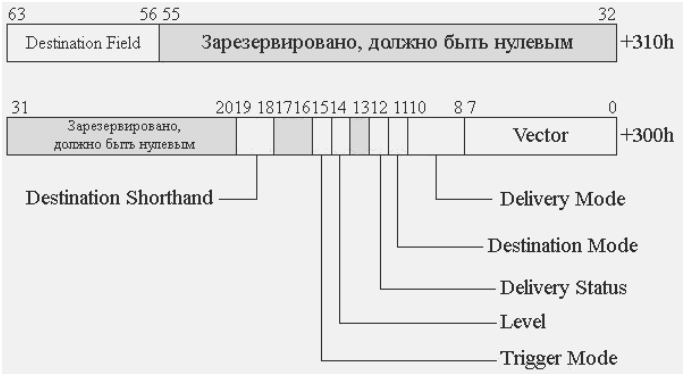


Рис. 6.8. Interrupt Command Register (ICR)

Interrupt Command Register – это 64-битный регистр, при записи в младшую часть которого генерируется межпроцессорное прерывание. Старшая часть имеет смещение 0310h от базового адреса Local APIC, младшая часть имеет смещение 0300h. Формат регистра ICR приведён на рис. 6.8.

Поле **Vector** (биты 0–7) содержит вектор прерывания, которое будет сгенерировано в целевых процессорах.

Поле **Delivery Mode** (биты 8–10) задаёт тип доставки прерывания. Существуют следующие типы доставки:

- Fixed (000) – вызывается обработчик прерывания, находящийся в указанном векторе;
- Lowest Priority (001) – отправляется процессорам, которые находятся в режиме низкого приоритета. Поддерживается не всеми процессорами;
- SMI (010) – прерывание доставляется как специальный сигнал SMI (обычно посылается перед переходом в режим энергосбережения). Вектор должен быть равен нулю (для совместимости с будущими программами);
- NMI (100) – прерывание доставляется как немаскируемое (Non Masked Interrupt). Если процессор находится в защищённом или 64-битном режиме, то будет вызван второй вектор из IDT. Номер вектора игнорируется;
- INIT (101) – отправка сигнала INIT процессорам (подробнее об этом типе прерывания будет рассказано позже). Вектор должен быть равен нулю (для совместимости с будущими программами);
- Start-Up (110) – специальный тип прерывания, иногда называемый SIPI, посылается процессорам, которые ещё не запущены. Вектор обычно указывает на код, который производит начальную инициализацию процессора и переводит его в нужный операционной системе режим;
- Значения 011 и 111 являются зарезервированными, и использовать их нельзя.

Поле **Destination Mode (**бит 11**)** задаёт тип адресата. Если в этом поле ноль, то тип адресата будет физическим; поле Destination Field будет содержать Local APIC ID целевого процессора. Если в поле Destination Mode единица, то режим

адресата будет логическим и адресат должен быть задан специальным образом. Подробнее логический режим адресата будет рассмотрен ниже.

Поле **Delivery Status** (бит 12) предназначено только для чтения и указывает статус межпроцессорного прерывания: ноль – если успешно доставлено, единица – если ещё в процессе.

Поле **Level** (бит 14) не используется в процессорах Pentium 4 и Xeon. Для сигнала INIT должно быть равно нулю, для всех остальных – единице.

Поле **Trigger Mode** (бит 15) используется только для типа сигнала INIT. Ноль – если нужна чувствительность к фронту сигнала; единица – к уровню сигнала.

Поле **Destination Shorthand** (биты 18–19) позволяет выбрать адресатов межпроцессорного прерывания:

- No Shorthand (00) – отправление сигнала процессору или процессорам, указанным в поле Destination Field;
- Self (01) – отправление сигнала о прерывании самому себе;
- All including self (10) – отправление сигнала о прерывании всем процессорам в системе, включая отправляющий;
- All excluding self (11) – отправление сигнала о прерывании всем процессорам в системе кроме отправляющего.

Поле **Destination Field** (биты 56–63) используется, только если поле Destination Shorthand содержит нули. Если выбрать физический тип адресата, то это поле должно содержать Local APIC ID целевого процессора; если выбран логический тип адресата, поле зависит от регистров LDR (Local destination register) и DFR (Destination format register).

У каждого Local APIC есть регистры LDR и DFR. Регистр LDR имеет смещение 0D0h от базового адреса Local APIC (по умолчанию 0FEE000D0h). Регистр задаёт логический идентификатор локального APIC. Биты 24–31 регистра LDR задают логический идентификатор локального APIC; остальные являются зарезервированными. Регистр DFR имеет смещение 0E0h от базового адреса Local APIC (по умолчанию 0FEE000E0h). В регистре DFR значащими являются только четыре старших бита (28–31). Биты 28–31 в регистре DFR могут принимать следующие значения: 1111 и 0000. При установке единиц в старших четырёх битах регистра DFR используется плоская модель адресации, а при установке нулей – кластерная модель. Последняя в данной книге рассматриваться не будет, т. к. требует дополнительных контроллеров на материнской плате.

При использовании плоской логической модели адресации каждый Local APIC, получая межпроцессорное прерывание, использует свой логический идентификатор и поле Destination Field полученного сообщения. Local APIC производит операцию AND с использованием своего логического идентификатора и полученным логическим адресом. Если результат операции будет ненулевым значением (будет выставлен хотя бы один бит), то Local APIC принимает прерывание на обработку.

6.2.3. Синхронизация доступа к данным

При работе с несколькими процессорами возникает проблема синхронизации доступа к общим данным. Результат одновременного несинхронизированного доступа

к одной переменной сразу нескольких процессоров всегда непредсказуем. Самый простой пример возникновения такой ситуации – это функция, которая отвечает за ведение лога. Есть переменная, которая содержит указатель на конец лога; начиная с него функция записывает новые данные в лог, после чего обновляет указатель конца лога. При несинхронизированном ведении лога несколькими процессорами в то время как один процессор добавляет данные в лог, второй процессор считывает старый указатель на конец лога и тоже начинает запись. В итоге данные, которые были добавлены первым процессором, стираются. Синхронизация доступа к данным является ключевым аспектом в разработке программного обеспечения, работающего как в многопроцессорных системах, так и в многозадачных операционных системах.

Синхронизация доступа к данным не менее актуальна и на многозадачных системах, которые работают на одном процессоре. При синхронизации доступа нескольких процессоров к общим данным возникает понятие *критической секции*. Критическая секция – это такой участок кода, который в некоторый момент времени должен выполняться только на одном процессоре. Данное определение критической секции не очень точное, т. к. очень часто используется многозадачная операционная система, где основной исполнительной единицей является задача (поток или процесс). Критическая секция – это такой участок кода, который в некоторый момент времени может выполнять только один поток.

Фундаментальный принцип синхронизации доступа к общим данным основан на использовании так называемых семафоров. Синхронизация на основе семафоров очень похожа на регулирование на железнодорожных путях. К примеру, есть участок дороги, по которому в одну из двух сторон может ехать только один поезд. С обеих сторон одноколейного участка дороги стоят семафоры. Если горит зелёный свет, то участок свободен; если же красный, то по участку едет поезд. Когда горит зелёный свет, то поезд, взъезжающий на одноколейный участок дороги, переключает свет на красный, тем самым блокируя выезд других составов на данный участок. Миновав одноколейный отрезок пути, машинист снова включает зелёный свет семафора.

Если перевести всё вышесказанное на компьютерный язык, то семафору можно уподобить некую переменную (вернее, какой-либо бит в ней). Но одной только переменной недостаточно для синхронизации. Нужен механизм, который позволит кратковременно блокировать доступ других процессоров к некоторой ячейке памяти. Именно для этого разработчиками компании Intel был введён префикс `LOCK`.

Префикс `LOCK` может быть использован со следующими инструкциями:

1. Команды работы с битами: BTR, BTS и BTC.
2. Команды обмена данными: XADD, CMPXCHG и CMPXCHG8B.
3. Унарные арифметически и логические операции: INC, DEC, NOT и NEG.
4. Бинарные арифметически и логические операции: ADD, ADC, SUB, SBB, AND, OR и XOR.
5. Префикс `LOCK` всегда применяется к инструкции XCHG, хоть и не указывается явно.

Использование префикса LOCK с любыми другими командами (либо с указанными командами, но без операции записи в память) приводит к генерации исключения #GP либо к исключению #UD. При указании префикса LOCK доступ к ячейкам памяти, с которыми работает команда, для других процессоров блокируется.

Итак, теперь, «вооружившись» префиксом LOCK, можно реализовать процедуры входа и выхода из критических секций. Код входа в критическую секцию будет таков (роль семафора исполняет нулевой бит переменной SEMAPHORE):

```
@@:
            bt      dword [SEMAPHORE], 0
            jnc     @b
            lock btr dword [SEMAPHORE], 0
            jnc     @b
```

Перед тем как войти в критическую секцию кода, нам надо дождаться, когда нулевой бит переменной SEMAPHORE станет равным единице. За это отвечают первые две команды – BT и JNC. В цикле проверяется содержимое нулевого бита переменной SEMAPHORE – пока он равен нулю (команда BT помещает указанный бит во флаг CF), цикл продолжается. Как только бит принимает ненулевое значение, происходит его сброс с помощью команды BTR. Перед сбросом бита она сначала заносит его значение во флаг CF. После выполнения операции BTR необходимо убедиться в том, что перед сбросом бит имел ненулевое значение. Если он был равен нулю, т. е. другой процессор успел сбросить его раньше, чем это сделали мы, то производится возврат к ожиданию. Если всё прошло успешно, то далее должна выполняться критическая секция кода.

Для выхода из критической секции необходимо просто сбросить нулевой бит в переменной SEMAPHORE. Производится это действие следующей командой:

```
lock bts dword [SEMAPHORE], 0
```

Вышеуказанный приём также называется *спин-блокировкой*; он является основополагающим при синхронизации доступа к общим данным.

При разработке приложений, работающих под современными операционными системами (Windows, Linux, BSD и др.), нет надобности в реализации подобных механизмов, т. к. операционная система предоставляет целый набор функций для синхронизации общего доступа к данным.

6.2.4. Инициализация многопроцессорной системы

В многопроцессорной системе всегда есть два типа процессоров: BSP (bootstrap processor) и AP (application processor). При включении многопроцессорной системы оборудованием динамически выбирается процессор, который будет включён и запущен; он будет являться BSP-процессором. Именно на BSP-процессоре выполняется BIOS, которая в свою очередь запускает операционную систему. Все остальные процессоры «спят» до получения межпроцессорного прерывания Start-up IPI (SIPI).

При включении, а точнее при получении прерывания INIT, процессор анализирует свой флаг BSP (восьмой бит в регистре IA32_APIC_BASE): если он выставлен,

то процессор переходит к выполнению кода BIOS; если сброшен – к ожиданию сигналов Start-up IPI.

Полное «пробуждение» процессора происходит в результате получения последовательности межпроцессорных прерываний INIT–SIPI–SIPI. Но этого недостаточно, т. к. после включения процессор находится в режиме реальных адресов и необходимо переключить каждый процессор в нужный операционной системе режим (защищённый или 64-битный).

Итак, алгоритм инициализации многопроцессорной системы следующий:

1. После включения BSP-процессор переходит в нужный операционной системе режим и глобально включает APIC (бит 11 в регистре IA32_APIC_BASE).
2. BSP-процессор посылает прерывание INIT всем процессорам, исключая себя. После получения сигнала INIT процессоры анализируют свой флаг BSP, а поскольку они все являются AP-процессорами, то он у них у всех будет сброшен и каждый из них перейдёт к ожиданию сигналов SIPI. После отправки межпроцессорного прерывания INIT необходимо подождать 10 мс перед следующим шагом.
3. BSP процессор посылает прерывание Start-up IPI всем процессорам, исключая себя. В качестве вектора необходимо указать номер страницы, на которой расположен код, инициализирующий AP-процессоры. После отправки первого Start-up IPI необходимо подождать 200 мс перед следующим шагом.
4. BSP-процессор повторно посылает прерывание Start-up IPI, отправленное им на предыдущем шаге всем процессорам, исключая себя.

Важным моментом в данном алгоритме является вектор, указанный при посылке прерывания Start-up IPI. При получении прерывания Start-up IPI процессор производит сдвиг полученного вектора влево на 12 бит (умножение на 1000h) и передаёт управление по полученному после операции сдвига адресу. Код, инициализирующий AP-процессоры, должен находиться по адресу, кратному 1000h (4096) байтам (т. е. быть выровненным на границу страницы).

6.2.5. Практика

Итак, все необходимые знания для написания программы, инициализирующей многопроцессорную систему, у нас уже есть. Осталось только создать эту программу.

Программа будет написана «с нуля» без использования каркаса из главы 2. Назначение программы (по-прежнему оформленной в виде COM-файла под DOS) – в том, чтобы выводить Local APIC ID каждого процессора, а в конце вывести суммарное количество процессоров.

Код, переводящий процессор в защищённый режим, будет для всех процессоров один. Поскольку код, инициализирующий AP-процессоры, должен начинаться с адреса, кратного 4 Кб, то его заранее (в начале программы) надо переместить на адрес, кратный 4 Кб. В листинге 6.7 приведён код, который производит перемещение всего кода программы на адрес, отвечающий нашим требованиям.

Листинг 6.7. Перемещение инициализирующего кода по адресу, кратному 4 Кб

```
   mov ax,3
   int 10h

   mov si, CODE_START
   xor ax, ax
   mov es, ax
   mov di, CODE_BASE_ADDRESS
   mov cx, PROGRAM_END-PROGRAM_START
   rep movsb

   mov ds, ax
   mov ss, ax
   mov es, ax
   jmp 0000:CODE_BASE_ADDRESS
```

В листинге 6.8 приведён код, который производит переключение процессора в защищённый режим – этот код один для всех процессоров в многопроцессорной системе. В отличие от программы, которая была написана в главе 2, вычислять линейный адрес GDT и адрес перехода в защищённый режим не требуется, т. к. все данные уже перемещены на нужные адреса.

Листинг 6.8. Перевод процессора в защищённый режим

```
CODE_START:
ORG CODE_BASE_ADDRESS
PROGRAM_START:
   in   al,92h
   or   al,2
   out    92h,al

   lgdt   fword [GDTR]

   cli
   in   al,70h
   or   al,80h
   out    70h,al

   mov    eax,cr0
   or   al,1
   mov    cr0,eax

   jmp CODE_SELEKTOR:PROTECTED_MODE_ENTRY_POINT
```

После перехода в защищённый режим производится перезагрузка всех регистров, инициализация стека и увеличение переменной-счётчика количества процессоров. После указанных действий выводится сообщение о том, что процессор в защищённом режиме (листинг 6.9).

Листинг 6.9. Инициализация семафора и подсчёт количества процессоров

```
use32

PROTECTED_MODE_ENTRY_POINT:
   mov ax, DATA_SELEKTOR
   mov ds, ax
   mov es, ax
   mov ss, ax

@@:
   bt dword [SEMAPHORE], 0
   jnc  @b
   lock btr dword [SEMAPHORE], 0
   jnc  @b
;---------------------
      inc dword [CPU_Count]
      mov esp, [CPU_Count]
      mov ebp, esp      ; save current CPU Count in ebp
      shl esp, 12
      add esp, STACK_BASE_ADDRESS
;---------------------
   lock bts dword [SEMAPHORE], 0

   mov esi, message1
   mov al, 0
   mov ah, 0
   mov bl, «5»
   call OutText
```

Следует отметить, что если бы производилась только операция инкремента переменной `CPU_Count`, то оформлять её как критическую секцию было бы необязательно, т. к. инструкцию `INC` можно применить с префиксом `LOCK` и никаких конфликтов возникнуть не должно. В нашем случае необходимо получить количество процессоров в системе, подсчитанное на текущий момент, чтобы стек не пересекался со стеками других процессоров (нельзя забывать, что память общая для всех процессоров). При выполнении кода, вычисляющего адрес стека и обновляющего переменную, которая содержит количество процессоров – а именно в промежутке между командами `inc [CPU_Count]` и `mov esp, [CPU_Count]` – необходимо, чтобы никакой другой процессор не смог изменить переменную CPU_Count.

Теперь, после обновления счётчика количества процессоров, необходимо включить APIC, получить его ID и вывести его на экран (листинг 6.10).

Листинг 6.10. Вывод APIC ID на экран

```
   mov ecx, IA32_APIC_BASE
   rdmsr
   bts eax, 11
   wrmsr
```

```
    mov edx, eax

    mov eax, [LOCAL_APIC_ID_REG_DEF]
    shr eax, 24

    mov esi, LAPICID
    mov ebx, 16
    call dword_to_STR

    mov esi, message2
    mov al, 0
    mov ebx, ebp
    mov ah, bl
    mov bl, "5"
    call OutText
```

При выводе строки с Local APIC ID используется значение из регистра EBP, который содержит количество процессоров, подсчитанное на момент обновления переменной `CPU_Count` (листинг 6.9). По сути, это число является порядковым номером процессора в системе. Оно необходимо для того, чтобы строки с Local APIC ID не перекрывали друг друга на экране.

После вывода своего идентификатора Local APIC надо проверить, является ли данный процессор BSP-процессором. Если да, то необходимо разбудить остальные процессоры последовательностью межпроцессорных прерываний `INIT–SIPI–SIPI`.

Листинг 6.11. Инициализация AP-процессоров

```
    mov ecx, IA32_APIC_BASE
    rdmsr
    bt eax, 8
    jnc .END

;boorstrap code
    mov  eax, INIT_IPI_command
    mov [APIC_ICR_low_DEF], eax
   @@:
    bt  dword [APIC_ICR_low_DEF],12
    jc  @b
    mov eax, 10000
    call delay


    mov  eax, STARTUP_IPI_command
    mov [APIC_ICR_low_DEF], eax
   @@:
    bt  dword [APIC_ICR_low_DEF],12
    jc  @b

    mov eax, 200
    call delay
```

```
 mov  eax, STARTUP_IPI_command
 mov [APIC_ICR_low_DEF], eax
@@:
 bt  dword [APIC_ICR_low_DEF],12
 jc  @b

 mov eax, 50000
 call delay

 mov eax, [CPU_Count]
 mov esi, CPU_Count_Str
 mov ebx, 10
 call dword_to_STR

 mov esi, message3
 mov ebx, eax
 inc ebx
 mov al, 0
 mov ah, bl
 mov bl, "5"
 call OutText
```

Функция `delay` производит ожидание на время (в микросекундах), указанное в регистре EAX.

После отсылки сигналов `INIT-SIPI-SIPI` все остальные процессоры начнут выполнять код начиная с метки `CODE_BASE_ADDRESS`. Другие процессоры произведут те же действия, что и BSP-процессор, за исключением того что они уже не будут посылать последовательность сигналов `INIT-SIPI-SIPI`.

Полный код программы находится на диске, прилагающемся к этой книге (см. архив, относящийся к данному разделу).

Наиболее удобная программа для эмуляции многопроцессорной системы – QEMU. Для эмуляции многопроцессорной системы необходимо запустить QEMU с флагом `-smp N`, где `N` – количество процессоров, например: `qemu.exe -fda A: -smp 8`.

6.2.6. Резюме

В данном разделе нами был изучен механизм межпроцессорного взаимодействия, а также алгоритм инициализации многопроцессорной системы. Для закрепления полученных знаний была написана программа, инициализирующая многопроцессорную систему.

ПРИЛОЖЕНИЯ

Приложение A. MSR-регистры

В данном приложении представлено описание MSR-регистров, которые упоминались в данной книге. Названия регистров приводятся в соответствии с документацией компании Intel.

Регистры, описанные в данном приложении, поддерживаются всеми процессорами, начиная с семейства Pentium 4, и будут поддерживаться всеми новейшими моделями процессоров для обеспечения обратной совместимости.

Процессоры Intel и AMD имеют ряд дополнительных флагов в некоторых регистрах. В данном приложении указаны только те биты и флаги, которые являются общими (или имеют общее назначение) для процессоров обоих производителей.

A.1. Регистр IA32_EFER

Регистр IA32_EFER (или EFER) имеет индекс 0C0000080h. Является одним из важнейших MSR-регистров, т. к. управляет переключением процессора в 64-битный режим и включением основных его возможностей.

Рис. A.1. Формат регистра IA32_EFER

Бит SCE разрешает использование инструкций SYCALL/SYSRET. Если данный бит сброшен, то попытка выполнить инструкцию вызовет исключение общей защиты (#GP).

Бит LME отвечает за переключение процессора в long mode. Если присвоить этому биту единицу, то процессор переключится в long mode.

Бит LMA является индикатором long mode. Доступен только для чтения. Если данный бит выставлен, то процессор работает в long mode.

Бит NXE разрешает использование бита NX в элементах таблиц, описывающих виртуальные страницы. Если данный бит сброшен, то использование бита NX приведёт к исключению общей защиты (#GP).

A.2. Регистры, используемые командами SYSENTER/SYSEXIT

Регистр IA32_SYSENTER_CS расположен по индексу 174h. В младших 16 битах содержит селектор сегмента кода обработчика команды SYSENTER. Остальные биты (16–63) зарезервированы.

Регистр IA32_SYSENTER_ESP расположен по индексу 175h. Содержит адрес верхушки стека обработчика команды SYSENTER.

Регистр IA32_SYSENTER_EIP расположен по индексу 176h. Данный регистр содержит адрес обработчика команды SYSENTER.

A.3. Регистры, используемые командами SYSCALL/SYSRET

Регистр IA32_STAR расположен по индексу 0C0000081h. Регистр содержит базовые значения для селекторов, используемых командами SYSCALL/SYSRET.

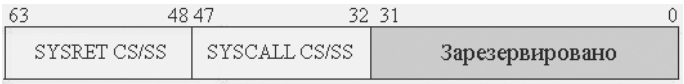


Рис. A.2. Формат регистра IA32_STAR

Биты 32–47 содержат значение, используемое при вычислении селектором целевого сегмента кода и стека при выполнении команды SYSCALL. Биты 32–47 содержат значение, используемое при вычислении селектором целевого сегмента кода и стека при выполнении команды `SYSRET`.

Регистр IA32_STAR расположен по индексу 0C0000082h. Регистр содержит 64-битный адрес обработчика команды SYSCALL.

Регистр IA32_FMASK расположен по индексу 0C0000084h. Регистр в младших 32 битах (старшие 32 бита зарезервированы) содержит маску для регистра флагов. При выполнении команды SYSCALL значение регистра флагов вычисляется путём выполнения операции AND текущего значения регистра флагов со значением из этого регистра.

A.4. Регистры APIC

Регистр IA32_APIC_BASE расположен по индексу 1Bh. Этот регистр позволяет задать базовый адрес регистров Local APIC в памяти.

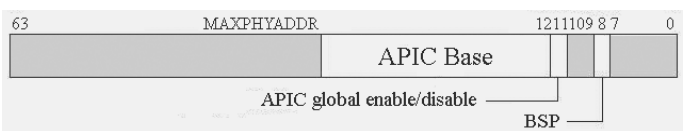


Рис. A.3. Формат регистра IA32_APIC_BASE

Поле APIC Base содержит базовый адрес регистров APIC. Базовый адрес всегда выровнен на границу страницы, поэтому значение младших 12 бит не имеет значения. После включения или перезагрузки процессора базовый адрес равен 0FEE00000h. Максимальная разрядность адреса зависит от модели процессора, но не может быть более 52.

Бит APIC global enable/disable отвечает за глобальное включение/выключение контроллеров APIC (Local APIC и IO APIC). Единица в этом бите включает контроллер, ноль – выключает контроллер.

Бит BSP показывает, является ли текущий процессор BSP-процессором. Если данный бит выставлен, то текущий процессор был первым процессором, который начал работать после включения.

А.5. Регистры для управления сегментами в long mode

Регистр IA32_FS_BASE расположен по индексу 0C0000100h. Этот регистр позволяет задать базовый адрес сегмента, описываемого регистром FS.

Регистр IA32_GS_BASE расположен по индексу 0C0000101h. Этот регистр позволяет задать базовый адрес сегмента, описываемого регистром GS.

Регистр IA32_KERNEL_GSBASE расположен по индексу 0C0000102h. Этот регистр используется командой SWAPGS для обмена значением с регистром IA32_GS_BASE.

А.6. Вспомогательные регистры

Регистр IA32_TSC_AUX расположен по индексу 0C0000103h. Младшие 32 бита используются командой RDTSCP, старшие 32 бита зарезервированы. Команда RDTSCP заносит значение этого регистра (младшие 32 бита) в ECX. Значение регистра может быть произвольным. Обычно он используется для передачи программам, работающим на ненулевых уровнях привилегий, информации о процессоре или другой вспомогательной информации.

Приложение Б. Системные регистры

В данном приложении приводится полное описание всех системных регистров, а также регистра флагов.

Процессоры Intel и AMD имеют ряд дополнительных флагов в некоторых регистрах. Ниже будут описаны только те биты и флаги, которые являются общими (или имеют общее назначение) для процессоров обоих производителей.

Б.1. Регистр CR0

Регистр CR0 управляет важнейшими параметрами работы процессора, такими как переход в защищённый режим, включение/выключение механизма трансляции страниц и др. Формат регистра CR0 приведён на рис. Б.1.

В защищённом режиме размер регистра составляет 32 бита, в 64-битном режиме регистр расширяется до 64 бит. Поля, помеченные серым цветом, являются зарезервированными. Зарезервированные поля всегда должны быть равны нулю – запись ненулевого значения в них приведёт к генерации исключения общей защиты (#GP).

Рис. Б.1. Формат регистра CR0

Бит PE (Protection Enable) отвечает за включение защищённого режима. Если бит равен единице, то процессор работает в защищенном режиме; если ноль, то в режиме реальных адресов.

Бит MP (Monitor Coprocessor) используется совместно с флагом TS и контролирует выполнение инструкции WAIT (FWAIT). Если флаги MP и TS выставлены, то при выполнении инструкции WAIT будет генерироваться исключение недоступного устройства (#NM). Если флаг MP сброшен, то команда WAIT никаких исключений генерировать не будет.

Бит EM (Emulation) показывает, существует ли математический сопроцессор. Если этот бит выставлен, то математического сопроцессора x87 нет либо он не подсоединён к процессору. Если этот бит выставлен, то все инструкции математического сопроцессора x87 будут генерировать исключение недоступного устройства (#NM). Таким образом, этот бит позволяет эмулировать программно выполнение инструкций x87. Также если бит EM выставлен, то все инструкции MMX и SSE будут генерировать исключение неверного опкода (#UD).

Бит TS (Task Switched) позволяет сохранять контекст математического сопроцессора, устройства MMX и SSE. Если этот флаг выставлен, то при выполнении любой инструкции математического сопроцессора x87, инструкции MMX или инструкции SSE будет генерироваться исключение недоступного устройства (#NM). Если этот флаг выставлен и выставлен флаг MP, то исключение недоступного устройства будет генерировать ещё и при выполнении инструкции WAIT (FWAIT).

Флаг TS выставляется самим процессором при любом переключении задачи. Таким образом, при помощи бита TS можно реализовать сохранение контекстов устройств x87, MMX и SSE, т. к. при переключении задач сам процессор не сохраняет их состояние в TSS.

Бит ET (Extension Type) в настоящее время зарезервирован и не используется – он всегда имеет значение 1. Данный флаг использовался только на процессорах i386 и i486 для определения поддержки набора инструкций Intel 387 DX.

Бит NE (Numeric Error) используется для выбора режима обнаружения ошибок математического сопроцессора x87. Если этот флаг равен единице, то включается внутренний (встроенный) механизм обработки исключений математического сопроцессора x87. Если флаг равен нулю, то сигналы об исключениях в математическом сопроцессоре контролируются сигналом #IGNNE. Если входной сигнал #IGNNE равен единице, то все исключения математического сопроцессора будут игнорироваться. Если сигнал #IGNNE равен нулю, то оповещение об исключении выполняется через вход #FERR.

Бит WP (Write Protect) включает защиту от записи для кода, выполняющегося на нулевом уровне привилегий. Если этот бит выставлен, то код, выполняющийся на нулевом уровне привилегий, не сможет производить операцию записи на страницы, помеченные «только для чтения». Если флаг WP сброшен (это вариант по умолчанию), то код, выполняющийся в нулевом кольце, может выполнять запись на любые страницы, даже на те, которые помечены «только для чтения».

Бит AM (Alignment Mask) управляет проверкой выравнивания. Проверка выравнивания включается после выполнения трёх условий: бит AM в регистре CR0

выставлен, бит AC в регистре флагов – тоже выставлен и текущий уровень привилегий равен трём. В случае выполнения всех условий при использовании невыровненных адресов будет генерироваться исключение проверки выравнивания (#AC).

Бит NW (Not Writethrough) игнорируется на большинстве современных процессоров.

Бит CD (Cache Disable) глобально включает/отключает механизм кэширования для процессора. Если этот бит сброшен, то кэширование данных работает в полную силу; если выставлен, то кэширование отключается и никаких новых данных в кэш заноситься не будет. Но даже при отключённом кэшировании процессор будет обращаться к кэшу в следующих случаях:

1. При операции чтения, если произошло кэш-попадание (искомые данные найдены в кэше), данные будут считаны из кэша.
2. При операции записи, если произошло кэш-попадание, производится операция записи в память и искомые данные в кэше помечаются как недействительные.

Также при отключённом кэшировании процессор будет игнорировать флаги PWT и PCD в структурах, описывающих страницы памяти (таблица страниц, каталог страниц и т. д.).

Бит PG (Paging) включает/отключает механизм трансляции адресов. Если этот бит выставлен, то механизм трансляции адресов включается; если сброшен – выключается. Механизм трансляции адресов нельзя включить, находясь в режиме реальных адресов, а также в других ситуациях, когда флаг PE сброшен.

Б.2. Регистры CR2 и CR3

Регистры CR2 и CR3 используются после включения механизма трансляции адресов.

Регистр CR2 актуален только после вызова обработчика страничного нарушения. Регистр CR2 содержит виртуальный адрес, из-за которого произошло

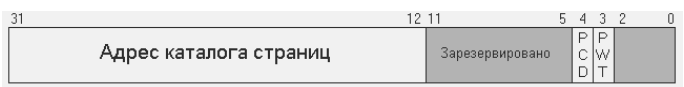


Рис. Б.2. Формат регистра CR3 в защищённом режиме (обычный режим трансляции адресов)

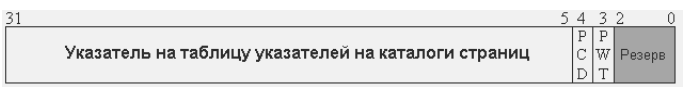


Рис. Б.3. Формат регистра CR3 в защищённом режиме (режим PAE)

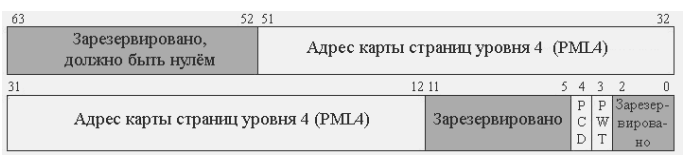


Рис. Б.4. Формат регистра CR3 long mode

страничное нарушение. В защищённом режиме регистр имеет размер 32 бита, в 64-разрядном режиме расширяется до 64 бит.

Регистр CR3 служит для трансляции виртуальных адресов в физические; он содержит адрес корневой таблицы (каталог страниц, таблица указателей на каталоги страниц и карта страниц четвёртого уровня), используемой при трансляции адресов.

Биты PCD и PWT управляют кэшированием страницы, на которой находится каталог страниц, таблица указателей на каталоги страниц или карта страниц четвёртого уровня. Флаг PCD отключает кэширование для этой страницы, флаг PWT разрешает/запрещает сквозную запись на ней.

В защищённом режиме при включённом режиме PAE нет необходимости размещать таблицу указателей на каталоги страниц по адресу, выровненному на 4 Кб – достаточно выравнивания на 32 байта. Но, несмотря на выравнивание, биты PCD и PWT будут управлять кэшированием для всей страницы, на которой будет находиться таблица указателей на каталоги страниц.

Б.3. Регистр CR4

Регистр CR4 управляет дополнительными возможностями процессора, а также возможностями, которыми не управляет регистр CR0.

В защищённом режиме размер регистра составляет 32 бита, в 64-битном режиме регистр расширяется до 64 бит. Поля, помеченные серым цветом, являются зарезервированными. Зарезервированные поля всегда должны быть равны нулю – запись ненулевого значения в них приведёт к генерации исключения общей защиты (`#GP`).

Для процессоров компании Intel предусмотрено больше значащих битов в регистре CR4, управляющих дополнительными возможностями, которые поддерживаются только процессорами данного производителя. Эти флаги рассматриваться здесь не будут.

Бит VME (Virtual-8086 Mode Extensions) контролирует поддержку аппаратных возможностей обработки прерываний и исключений для программ, работающих в режиме виртуального процессора 8086.

Бит PVI (Protected-Mode Virtual Interrupts) включает/отключает поддержку флагов виртуальных прерываний в регистре флагов (VIF и VIP).

Бит TSD (Time-Stamp Disable) запрещает выполнение инструкций RDTSC и RDTSCP на ненулевом уровне привилегий. Если этот бит выставлен, то выполнять инструкции RDTSC и RDTSCP можно только на нулевом уровне привилегий; если сброшен – то на любом уровне привилегий.

Рис. Б.5. Формат регистра CR4

Бит DE (Debugging Extensions) запрещает обращение к регистрам DR4 и DR5. Если этот бит равен единице, то любое обращение к регистрам DR4 и DR5 вызовет исключение неизвестного опкода (#UD).

Бит PSE (Page Size Extensions) разрешает использование страниц размером 4 Мб. Данный флаг актуален только в обычном режиме трансляции адресов в защищённом режиме процессора.

Бит PAE (Physical Address Extension) включает режим расширенной физической адресации (размер физического адреса может превышать 32 бит). Данный флаг может быть включён только совместно с флагом PG в регистре CR0. Режим расширенной физической адресации – одно из обязательных требований для работы в long mode.

Бит MCE (Machine-Check Enable) разрешает исключение проверки оборудования (#MC). Если данный бит сброшен, то исключение проверки оборудования генерироваться не будет.

Бит PGE (Page-Global Enable) разрешает использование бита G в таблицах и каталогах страниц. Если данный бит равен нулю, то флаг G в таблицах и каталогах страниц будет игнорироваться.

Бит PCE (Performance-Monitoring Counter Enable) разрешает использование инструкции RDPMC программам, работающим на ненулевом уровне привилегий. Если этот флаг сброшен, то выполнять инструкцию RDPMC может только код, работающий на нулевом уровне привилегий.

Бит OSFXSR (Operating System Support for FXSAVE and FXRSTOR instructions) разрешает/запрещает выполнять набор инструкций SSE. Если данный флаг равен единице, то инструкции FXSAVE и FXRSTOR могут сохранять и загружать состояние регистров x87, MMX и SSE; также разрешается использование набора инструкций SSE. Если этот бит сброшен, то инструкции FXSAVE и FXRSTOR могут сохранять и загружать только состояние регистров x87 и MMX, и любая инструкция SSE будет генерировать исключение неизвестного опкода (#UD).

Бит OSXMMEXCPT (Operating System Support for Unmasked SIMD Floating-Point Exceptions) разрешает генерацию исключения плавающей точки в SIMD (#XF). Если данный бит равен единице, то при возникновении ошибок в работе инструкций SSE будет генерироваться исключение плавающей точки в SIMD (#XF). Если данный бит сброшен, то при возникновении ошибок в работе с плавающей точкой будет генерироваться исключение неизвестного опкода (#UD).

Б.4. Регистры GDTR и IDTR

Регистры GDTR и IDTR служат для указания параметров глобальных таблиц: глобальной дескрипторной таблицы (GDT) и глобальной таблицы прерываний (IDT). Формат регистров GDTR и IDTR приведён на рис. Б.6.

Рис. Б.6. Формат регистров GDTR и IDTR

Размерность регистров в защищённом режиме – 48 бит; в long mode регистры расширяются до 80 бит для возможности хранения в них 64-битных адресов таблиц.

Поле лимита таблицы задаётся максимальное допустимое смещение внутри неё, которое равно размеру таблицы в байтах минус 1. Адрес таблицы является абсолютным виртуальным адресом, он не будет зависеть от содержимого каких-либо сегментных регистров.

Б.5. Регистры LDTR и TR

Регистры LDTR и TR имеют размер 16 бит и хранят селекторы локальной дескрипторной таблицы (LDT) и текущей выполняющейся задачи. Также в этих регистрах есть теневые части, которые содержат сами дескрипторы в целях минимизации обращений к глобальной дескрипторной таблице (GDT).

В long mode регистр TR служит лишь для указания стеков (чтобы можно было обеспечить переключение стеков при переходе с одного уровня привилегий на другой), а также для указание IST-фреймов.

Б.6. Регистр флагов

Регистр флагов является ключевым в работе процессора, поскольку управляет основополагающими аспектами его работы.

В защищённом режиме регистр флагов имеет размер 32 бита, в long mode расширяется до 64 бит.

Флаг CF (Carry flag) – флаг переноса. Он выставляется процессором, если имеет место перенос или заем из старшего бита результата. Флаг полезен для произведения операций над числами длиной в несколько слов, когда имеют место переносы и заёмы из слова в слово.

Флаг PF (Parity flag) – флаг чётности. Выставляется процессором всякий раз, если младший байт результата имеет чётное количество единичных битов.

Флаг AF (Adjust flag) – дополнительный флаг переноса. Используется во время выполнения команд десятичного сложения и вычитания при необходимости переноса или заёма между полубайтами.

Флаг ZF (Zero flag) – флаг нулевого результата. Выставляется всякий раз, когда результатом последней операции был ноль; в противоположном случае сбрасывается.

Флаг SF (Sign flag) – флаг знака. Всегда равен самому старшему биту результата последней операции. Де-факто является индикатором отрицательного результата. Если данный флаг равен единице, значит, результатом последней операции было отрицательное число.

Флаг TF (Trap Flag) – флаг трассировки. Если равен единице, то после выполнения каждой инструкции процессор будет генерировать отладочное исключение

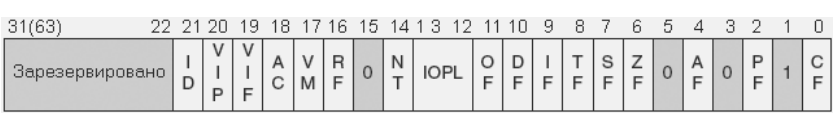


Рис. Б.7. Регистр флагов

(#DB). Если данный флаг выставляется после выполнения инструкций POPF или IRET, то отладочное исключение будет сгенерировано после выполнения следующей инструкции.

Флаг IF (Interrupt flag) – флаг запрещения прерываний. Если он выставлен, то будут запрещены все маскируемые аппаратные прерывания. Данный флаг никак не влияет на генерацию исключений и немаскируемые прерывания (NMI).

Флаг DF (Direction flag) – флаг направления. Управляет работой инструкций MOVS, CMPS, SCAS, LODS, OUTS и STOS. Если данный флаг выставлен, то инструкции работы со строками будут работать в обратную сторону, т. е. от старших адресов к младшим. Если флаг сброшен, инструкции работают в обычном режиме.

Флаг OF (Overflow flag) – флаг переполнения. Выставляется процессором всякий раз, когда возникает арифметическое переполнение, т. е. когда объем результата превышает размер ячейки назначения.

Поле IOPL (I/O privilege level) – уровень привилегий ввода/вывода. Для того чтобы текущая задача могла получить доступ к портам ввода/вывода (если он запрещён), её уровень привилегий должен быть численно меньше или равен IOPL. Данное поле можно изменить (команды POPF и IRET) только на нулевом уровне привилегий.

Флаг NT (Nested Task) – флаг вложенной задачи. Выставляется всякий раз, когда произошло переключение задачи командой CALL или генерацией прерывания либо исключения, обработчик которого является задачей. Если данный флаг установлен при выполнении команды IRET, произойдёт переключение на прерванную задачу.

Флаг RF (Resume flag) – флаг повтора. Флаг RF отвечает за рестарт инструкции, вызвавшей исключение; одновременно с этим данный флаг запрещает отладочное исключение (`#DB`). Для исключений типа «ловушка» не подразумевается рестарт инструкции, поэтому в стек обработчика помещается регистр флагов с флагом RF = 0. Процессор очищает этот флаг после выполнения каждой инструкции, за исключением двух случаев:

1. После выполнения команды IRET, которая установила данный флаг.
2. После команды JMP, CALL или INT n, если целью являлся шлюз задачи.

Также при генерации прерываний в стек обработчика помещается регистр флагов с флагом RF = 1, если прерванная инструкция являлась инструкцией работы со строками (MOVS, LODS, CMPS, SCAS, STOS и OUDS).

Флаг VM (Virtual-8086 Mode) – флаг виртуального процессора 8086. Для включения режима виртуального процессора 8086 надо установить значение этого бита в единицу. Для перехода обратно в защищённый режим необходимо сбросить значение данного бита. Флаг VM можно изменить только командой IRET.

Флаг AC (Alignment Check) – флаг проверки выравнивания. Если данный бит выставлен совместно с флагом AM в регистре CR0, то при использовании невыровненных адресов операндов на третьем уровне привилегий будет генерироваться исключение проверки выравнивания (`#AC`).

Флаг VIF (Virtual Interrupt flag) – виртуальный флаг IF. При установке флагов VME или PVI в регистре CR4 данный флаг является заменой флагу IF. Если выставлен флаг VME, то все изменения, происходящие с флагом IF в режиме виртуального процессора 8086, при IOPL меньше 3 будут отражаться на флаге VIF без возникновения исключения общей защиты (#GP). Если выставлен флаг PVI, то все изменения, происходящие с флагом IF в на третьем уровне привилегий, при IOPL меньше 3 будут отражаться на флаге VIF без возникновения исключения общей защиты (#GP).

Флаг VIP (Virtual interrupt pending) устанавливается в единицу, чтобы указать процессору, что виртуальное прерывание находится на обработке. Данный флаг никогда не модифицируется процессором и проверяется лишь при изменении битов VME или PVI в регистре CR4.

Флаг ID (Identification flag): способность программного обеспечения изменить этот бит указывает на то, что реализация процессора поддерживает машинную команду CPUID.

Б.7. Регистр CR8

Регистр CR8 является одним из новшеств long mode и предоставляет дополнительный способ доступа к регистру TPR Local APIC.

Он задаёт приоритет текущей выполняющейся задачи. Регистр имеет размерность 64 бита, но значимы в нём только младшие 4 – остальные зарезервированы и должны быть нулями. Младшие 4 бита задают уровень приоритета текущей задачи. Процессор будет принимать на обработку только те прерывания, которые численно больше, чем уровень приоритета текущей задачи. Установка нулевого значения в этом поле разрешит приём всех прерываний; установка значения 15 запретит приём всех прерываний.

Приложение В. Системные команды

В данном приложении приводится описание всех системных команд. К *системным командам* относятся все привилегированные команды, а также команды, работающие с системными регистрами и системными структурами данных.

Также у процессоров Intel и AMD есть дополнительные наборы команд (например, VT-x и AMD-V) – здесь они описаны не будут. Кроме того, надо отметить, что у процессоров обоих производителей возможны различия в работе некоторых команд. В данном приложении рассматриваются только те команды, которые поддерживаются всеми процессорами (и имеют одинаковые свойства).

В.1. Работа с системными регистрами

1. **LGDT** – загрузка регистра GDTR, указателя на глобальную дескрипторную таблицу. Синтаксис команды: `lgdt <источник>`. Источником может быть 48-битная (в защищённом и реальном режиме) или 80-битная (в 64-битном режиме) переменная, содержащая образ регистра GDTR. Инструкцию можно выполнить только на нулевом уровне привилегий, а также в режиме реальных адресов.

2. **SGDT** – получение содержимого регистра GDTR. Команда идентична `LGDT`, за исключением того что может быть выполнена на любом уровне привилегий.
3. **LIDT** – загрузка регистра IDTR, указателя на глобальную таблицу прерываний. Синтаксис команды: `lidt <источник>`. Источником может быть 48-битная (в защищённом и реальном режиме) или 80-битная (в 64-битном режиме) переменная, содержащая образ регистра GDTR. Инструкцию можно выполнить только на нулевом уровне привилегий, а также в режиме реальных адресов.
4. **SIDT** – получение содержимого регистра IDTR. Команда идентична LIDT, за исключением того что может быть выполнена на любом уровне привилегий.
5. **LLDT** – загрузка регистра LDTR. Синтаксис команды: `lldt <источник>`. Источником может быть 16-битный регистр или значение в памяти, содержащее селектор сегмента LDT в глобальной дескрипторной таблице. Инструкцию можно выполнить только на нулевом уровне привилегий.
6. **SLDT** – получение содержимого регистра LDTR. Команда идентична LLDT, за исключением того что может быть выполнена на любом уровне привилегий.
7. **LMSW** – загрузка машинного статуса. Синтаксис команды: `lmsw <источник>`. Источником может быть 16-битный регистр или значение в памяти. Команда производит загрузку младших 4 бит источника в младшие 4 бита регистра CR0. Инструкцию можно выполнить только на нулевом уровне привилегий. Она поддерживается лишь для совместимости со старым программным обеспечением.
8. **SMSW** – получение машинного статуса (младшие 16 бит регистра CR0). Синтаксис команды: `smsw <назначение>`. В качестве назначения может быть указан регистр любой размерности, результат будет расширен нулями до необходимого размера. Также в качестве назначения может быть указана 16-битная ячейка в памяти.
9. **CLTS** – сброс флага TS в регистре CR0. Команда не принимает параметров. Команду можно выполнить только на нулевом уровне привилегий.
10. **LTR** – загрузка регистра задачи. Синтаксис команды: `ltr <источник>`. Источником может быть 16-битный регистр или значение в памяти, содержащее селектор сегмента TSS в глобальной дескрипторной таблице. Инструкцию можно выполнить только на нулевом уровне привилегий.
11. **STR** – получение содержимого регистра задачи. Команда идентична LTR, за исключением того что может быть выполнена на любом уровне привилегий.
12. **MOV (CRn)** – работа с системными регистрами CRn. Синтаксис команды: `mov <источник>, <назначение>`. Если источником является системный регистр CRn, то назначением может быть только регистр общего назначения. Если назначением является системный регистр CRn, то источником может быть только регистр общего назначения. Команду можно выполнить

только на нулевом уровне привилегии, а также в режиме реальных адресов. В защищённом и реальном режимах размерность регистров – 32-бита, в 64-битном режиме – 64-бита.

13. **MOV (DRn)** – работа с отладочными регистрами. Синтаксис команды: `mov <источник>, <назначение>`. Если источником является отладочный регистр, то назначением может быть только регистр общего назначения. Если назначением является отладочный регистр, то источником может быть только регистр общего назначения. Команду можно выполнить только на нулевом уровне привилегий, а также в режиме реальных адресов. В защищённом и реальном режимах размерность регистров – 32-бита, в 64-битном режиме – 64-бита.
14. **RDMSR** – чтение из MSR-регистра. Инструкция не принимает параметров. Индекс регистра задаётся в регистре ECX, содержимое MSR-регистра сохраняется в регистрах EDX:EAX (старшая часть – в EDX, младшая – в EAX). Инструкция может быть выполнена только на нулевом уровне привилегий, а также в режиме реальных адресов.
15. **WRMSR** – запись в MSR-регистр. Инструкция не принимает параметров. Работа команды идентична RDMSR.
16. **RDPMC** – чтение регистров счётчика производительности. Инструкция не принимает параметров. Индекс регистра задаётся в регистре ECX (0–3), содержимое счётчика производительности сохраняется в регистрах EDX:EAX (старшая часть – в EDX, младшая – в EAX). Если бит PCE в регистре CR4 выставлен, инструкцию можно выполнять на любом уровне привилегий; если сброшен, то только на нулевом уровне. Команда равносильна обращению к MSR-регистрам, содержащим счётчики производительности, но их индексы на каждом типе процессоров индивидуальны. Команда RDPMC унифицирует обращение к MSR-регистрам, содержащим счётчики для всех процессоров.
17. **RDTSC** – чтение счётчика времени. Инструкция не принимает параметров. Содержимое счётчика производительности сохраняется в регистрах EDX:EAX (старшая часть – в EDX, младшая – в EAX). Если бит TSD в регистре CR4 сброшен, инструкцию можно выполнять на любом уровне привилегий; если выставлен, то только на нулевом уровне.
18. **RDTSCP** – чтение счётчика времени и дополнительного значения. Работа данной команды идентична работе команды RDTSC, за исключением того что в регистр ECX помещается содержимое младших 32 бит MSR-регистра IA32_TSC_AUX. Содержимое регистра IA32_TSC_AUX может быть произвольным и обычно используется для того, чтобы передать программам на ненулевых уровнях привилегий информацию о процессоре.
19. **SWAPGS** – обмен теневой части регистра GS. Команда не принимает параметров. Обмен производится со значением MSR-регистра IA32_KERNEL_GS_BASE. Новое значение базы для сегмента, описываемого регистром GS, загружается в теневую часть GS; старое значение базы GS заносится в регистр IA32_KERNEL_GS_BASE. Команда может быть выполнена только на нулевом уровне привилегий и только в 64-битном режиме.

B.2. Системные команды

1. **HLT** – перевод процессора в состояние HALT. Команда не принимает параметров. Состояние HALT характеризуется тем, что процессор не выполняет никаких инструкций и потребляет минимум энергии. Процессор выходит из состояния HALT в результате следующих событий: аппаратные прерывания (если они разрешены), немаскируемое прерывание (NMI), прерывание SMI, приём сигналов RESET или INIT. После выхода из состояния HALT будет выполнена инструкция, следующая за HLT. Данная команда может быть выполнена только на нулевом уровне привилегий.
2. **IRET (IRETD, IRETQ)** – возврат из прерывания. Команда не принимает параметров. Если происходит смена уровня привилегий, то происходит переключение стека (старые значения SS и ESP извлекаются из стека обработчика прерывания, в long mode SS и RSP всегда извлекаются из стека обработчика прерывания). Если выставлен флаг NT в регистре флагов, то в защищённом режиме происходит переключение задач, а в long mode генерируется исключение общей защиты (`#GP`).
3. **MONITOR** – инициализация и переход в состояние мониторинга. Инструкция не принимает параметров. Регистр EAX (RAX) должен содержать адрес памяти, изменения в котором будут отслеживаться, и в случае изменения в указанной ячейке памяти процессор будет выходить из состояния мониторинга. Регистры EDX и ECX должны содержать дополнительные параметры, но на современных процессорах они ещё не определены. Содержимое регистра EDX игнорируется полностью, а регистр ECX должен быть обнулён. Инструкция может быть выполнена только на нулевом уровне привилегий.
4. **MWAIT** – переход в состояние ожидания. Команда не принимает параметров. Регистры EAX и ECX задают дополнительные параметры ожидания. Регистр EAX полностью игнорируется. Нулевой бит в регистре ECX разрешает выход из состояния ожидания даже в случае аппаратных прерываний, в том числе и если флаг `IF` сброшен. Все остальные биты в регистре ECX должны быть равны нулю. Данную инструкцию можно выполнить только на нулевом уровне привилегий, а также в режиме реальных адресов.
5. **RSM** – выход из режима системного управления (SMM). Команда не принимает параметров. Инструкцию можно выполнить только в режиме системного управления: в любом другом режиме она вызовет исключение недействительного опкода (`#UD`).
6. **SYSENTER** – вызов системной функции. Инструкция не принимает параметров. Работает только в защищённом режиме. Производится переход на нулевой уровень привилегий. Новое значение регистра CS загружается из MSR-регистра IA32_SYSENTER_CS, новое значение SS вычисляется путём прибавления числа 8 к значению CS. В теневые части регистров CS и SS загружаются такие значения, базовые адреса которых равны нулю, а лимиты – 4 Гб. Новое значение ESP загружается из MSR-регистра

IA32_SYSENTER_ESP. Производится прыжок (загрузка регистра EIP) на адрес, указанный в MSR-регистре IA32_SYSENTER_EIP. В регистре флагов сбрасываются флаги VM, IF и RF.

7. **SYSEXIT** – выход из системной функции. Инструкция не принимает параметров. Работает только в защищённом режиме. Новое значение регистра CS вычисляется путём прибавления числа 16 к значению из MSR-регистра IA32_SYSENTER_CS; новое значение SS вычисляется путём прибавления числа 24 к значению из MSR-регистра IA32_SYSENTER_CS. В теневые части регистров CS и SS загружаются значения, базовые адреса которых равны нулю, а лимиты – 4 Гб. Всегда производится переход на третий уровень привилегий. Новое значение ESP загружается из регистра ECX. Производится прыжок (загрузка регистра EIP) на адрес, указанный в регистре EDX. Инструкцию можно выполнить только на нулевом уровне привилегий.
8. **SYSCALL** – вызов системной функции (64-битный режим). Инструкция не принимает параметров. Работает только в 64-битном режиме. Происходит переход на нулевой уровень привилегий. Новое значение CS загружается из MSR-регистра IA32_STAR (биты 32–47). Новое значение SS вычисляется путём добавления числа 8 к регистру CS. Происходит переход (загрузка нового значения в регистр RIP) по адресу, указанному в MSR-регистре IA32_LSTAR; адрес возврата сохраняется в регистре RCX. В регистр флагов загружается значение, вычисленное путём операции AND со значением из MSR-регистра IA32_FMASK. Старое значение регистра флагов сохраняется в регистре R11. Для включения поддержки данной инструкции необходимо установить в единицу бит SCE в MSR-регистре IA32_EFER.
9. **SYSRET** – выход из системной функции (64-битный режим). Инструкция не принимает параметров. Работает только в 64-битном режиме. Инструкцию можно выполнить только на нулевом уровне привилегий. Всегда происходит переход на третий уровень привилегий. Поддерживается возврат как в 64-битный режим, так и в режим совместимости. Если указан префикс `REX.W`, то возврат происходит в 64-битный режим; если же префикс не указан, возврат производится в режим совместимости. Новый регистр флагов загружается из регистра R11. Если возврат происходит в режим совместимости, то новое значение CS загружается из MSR-регистра IA32_STAR (биты 48–63); новое значение SS вычисляется путём прибавления числа 8 к значению из регистра CS. Если возврат происходит в 64-битный режим, то новое значение CS загружается из MSR-регистра IA32_STAR (биты 48–63) и к нему прибавляется число 16; новое значение SS загружается из MSR-регистра IA32_STAR (биты 48–63) и к нему прибавляется число 8. Происходит прыжок (загрузка нового значения в регистр RIP) на адрес, указанный в RCX (ECX – при возврате в режим совместимости). Для включения поддержки данной инструкции необходимо установить в единицу бит SCE в MSR-регистре IA32_EFER.

B.3. Работа с кэшем процессора

1. **INVD** – освобождение внутреннего кэша процессора. Команда не принимает параметров. Обратная запись в оперативную память не производится. Команда не затрагивает TLB-кэш. Может быть выполнена только на нулевом уровне привилегий.
2. **INVLPG** – удаление TLB-записи, ассоциированной с указанным адресом. Синтаксис команды: `invlpg <адрес>`. Команда может быть выполнена только на нулевом уровне привилегий.
3. WBINVD – освобождение внутреннего кэша процессора с обратной записью. Все модифицированные данные, находящиеся в кэше прцоессора, будут отображены на основную память. Команда не затрагивает TLB-кэш. Может быть выполнена только на нулевом уровне привилегий.

B.4. Дополнительные команды

1. **ARPL** – увеличение запрашиваемого уровня привилегий (RPL). Синтаксис команды: `arpl <назначение>, <источник>`. Первый параметр может быть 16-битным регистром или ячейкой памяти, второй – только 16-битным регистром. Команда ARPL сравнивает поля RPL (младшие 2 бита) назначения и источника: если поле RPL назначения численно меньше, чем источника, то полю RPL назначения присваивается RPL источника; также флаг ZF устанавливается в единицу. В противоположном случае флаг ZF сбрасывается. Команду можно выполнять на любом уровне привилегий, но фактически она полезна только на 1-м и 2-м уровнях. В 64-битном режиме не работает.
2. **CLI** – сброс флага IF. Команда не принимает параметров. Если текущий уровень привилегий больше, чем уровень привилегий ввода/вывода, то будет сгенерировано исключение общей защиты (`#GP`). Если в регистре CR4 выставлен флаг VME, при этом процессор находится в режиме виртуального 8086 и текущий уровень привилегий ввода/вывода (IOPL) меньше 3, то будет обнулён флаг VIF вместо IF и исключение общей защиты сгенерировано не будет. Если в регистре CR4 выставлен флаг PVI, текущий уровень привилегий равен 3 и текущий уровень привилегий ввода/вывода (IOPL) меньше 3, то будет обнулён флаг VIF вместо IF и исключение общей защиты сгенерировано не будет.
3. **STI** – установка флага IF в единицу. Инструкция идентична команде CLI.
4. **INT3** – вызов обработчика исключения точки останова (`#BP`) (короткий опкод). Команда не принимает параметров. От другой такой же команды отличается коротким однобайтовым опкодом (инструкция INT 3 занимает два байта), а также тем, что если в регистре CR4 выставлены биты VME или PVI, обработчик исключения точки останова (`#BP`) будет вызван вне зависимости от того, запрещён он или нет.
5. **LAR** – загрузка прав доступа дескриптора. Синтаксис команды: `lar <назначение>, <селектор>`. Селектор может быть указан как регистром, так и ячейкой памяти. Назначением может быть только регистр. Селектор может

указывать на дескриптор как в GDT, так и в LDT. Если в качестве назначения указан 16-битный регистр, то из дескриптора извлекается третье слово (байты 5 и 6), производится операция AND со значением 0FF00h и результат помещается в назначение. Если в качестве назначения указан 32-битный или 64-битный регистр, то из дескриптора извлекается второе двойное слово (5–8 байты), производится операция AND со значением 00FFFF00h и результат помещается в назначение.

6. **LSL** – загрузка лимита сегмента. Синтаксис команды: `lsl <назначение>, <селектор>`. Селектор может быть указан как регистром, так и ячейкой памяти. Назначением может быть только регистр. Селектор может указывать на дескриптор как в GDT, так и в LDT. Команда загружает лимит указанного сегмента в назначение. Лимит вычисляется с учётом бита гранулярности.
7. **UD2** – генерация исключения недействительного опкода (`#UD`). Инструкция не принимает параметров. Может быть выполнена в любом режиме и на любом уровне привилегий.
8. **VERR** – проверка возможности чтения из сегмента. Синтаксис команды: `verr <селектор>`. Селектор может быть указан как регистром, так и ячейкой памяти. Если текущий код сможет читать из сегмента, описываемого селектором, то флаг ZF установится в единицу, в противном случае будет сброшен.
9. **VERW** – проверка возможности записи в сегмент. Работа инструкции идентична работе VERR.

Алфавитный указатель

П

Р

С

Т

У

Ф

Х

Ш

Э

Я

A

B

C

D

E

F

G

H

I

L

M

O

P

R

S

T

U

V

W

X

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС-КНИГА» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Internet-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. **(495) 258–9194, 258–9195**; электронный адрес **books@alians-kniga.ru.**

Аблязов Руслан Зуфярович

Программирование на ассемблере на платформе x86-64

Главный редактор *Мовчан Д.А.*
dm@dmk-press.ru
Литературный редактор *Готлиб О.В.*
Верстка *Татаринов А.Ю.*
Дизайн обложки *Мовчан А.Г.*

Подписано в печать 02.02.2011. Формат 70×100 $^{1}/_{16}$.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 24,6. Тираж 1000 экз.
№

Электронный адрес издательства: **www.dmk-press.ru**