

Глава 7. Система прерываний

*То, что неясно, следует выяснить. То,
что трудно, следует делать с великой
настойчивостью.*

Конфуций, V век до н.э.

Я не люблю, когда – наполовину

Или когда прервали разговор.

В. Высоцкий

Далее продолжим изучение *переходов*, т.е. нарушений естественного порядка выполнения команд программы. Как уже упоминалось, некоторые переходы производятся не при выполнении команд программы, а могут делаться процессором *автоматически* при возникновении определенных условий. Если компьютер обладает такими возможностями, то говорят, что в нём реализована *система прерываний* (иногда её называют ещё *аппаратом прерываний*). Заметим, что самые первые компьютеры (их относят к ЭВМ первого и, частично, второго поколений) такими возможностями не обладали, однако все современные ЭВМ имеют систему прерываний, и сейчас Вам предстоит изучить, что это такое.

Сначала о терминологии. При запуске на счёт программа (её часто называют *приложение* – Application) образует *задачу* (task в ОС Windows) или *процесс* (process в ОС Unix), это синонимы, мы будем использовать термин *задача*. Разные программы порождают разные задачи, однако и одна программа может породить несколько независимых задач.¹ Сама по себе задача пассивна, для своего счёта она порождает один или несколько (вычислительных) *потоков* (в Windows) или *нитей* (в Unix)², будем использовать термин *поток*. Итак, каждая программа при запуске на счёт порождает хотя бы одну задачу, а каждая задача имеет хотя бы один поток. Поток является независимой единицей работы операционной системы (для диспетчера задач). Приложение завершается, когда завершаются все его задачи, а задача завершается, когда завершаются все её потоки. Итак, основным является понятие (вычислительного) потока, грубо говоря, это просто выполнение некоторой программы (команда за командой).

Сначала введём понятие *события* (возникшей ситуации), которое может касаться как выполняемого на ЭВМ потока (это *внутреннее* событие), так и функционирования периферийных устройств (это *внешнее* событие). Внутренние события могут возникать в процессоре и оперативной памяти (например, деление на ноль, попытка выполнить машинную команду с несуществующим кодом операции, сбой в работе оперативной памяти, выполнение некоторых особых команд и т.д.). Такие события часто называют *синхронными*, так как они возникают только при выполнении каких-то команд программы.

Внешние (*асинхронные*) события могут возникать в периферийных устройствах. Например, нажатая кнопка мыши, на печатающем устройстве кончилась бумага, пришел сигнал на сетевую карту, диск закончил обмен данными с оперативной памятью, понизилась температура процессора (можно уменьшить скорость вращения охлаждающего вентилятора) и др. Чаше всего бывает так, что при возникновении некоторого события продолжать выполнение текущего потока может быть либо бессмысленно (деление на ноль), либо нежелательно, так как нужно срочно предпринять какие-то действия, для выполнения которых этот поток просто не предназначен. Например, надо отреагировать на нажатие кнопки мыши, на сигнал от встроенного таймера, на сообщение, пришедшее по линии связи от другого компьютера, и т.д. Такие реакции на события реализуют специальные *процедуры обработки прерывания* (Interrupt Service Routine, Interrupt Handler).

Не надо думать, что события в компьютере – это нечто экзотическое, и они происходят редко. Действительно, некоторые из событий (например, деление на ноль) происходят нечасто, в то время как другие (например, события во встроенных в компьютере часах, которые говорят о том, что истек очередной квант времени) могут происходить несколько сот или даже тысяч раз в секунду.

¹ Например, браузеры (скажем, Chrome), порождает отдельную задачу для каждой открытой вкладки.

² Поток и нить (*жарг.* тред) – это синонимы, два разных перевода английского слова thread.

В архитектуре современных компьютеров предусмотрено, что каждое устройство, в котором произошло событие (процессор, оперативная память, устройства ввода/вывода) генерирует при этом особый *сигнал (запрос) прерывания* (Interrupt Request) – электрический импульс, который приходит на специальную электронную схему процессора. Сигнал прерывания, связанный с каждым из событий, обычно имеет свой *номер*, чтобы процессор мог отличить его от сигналов, связанных с другими событиями. По месту возникновения сигналы прерывания, как и события, классифицируются как *внутренние* (в процессоре и оперативной памяти) и *внешние* (в периферийных устройствах).

Получив сигнал прерывания, процессор автоматически (не выполняя никаких команд какой-либо программы!) предпринимает некоторые действия, которые называются *аппаратной реакцией* на этот сигнал. Надо сказать, что, хотя такая реакция, конечно, зависит от архитектуры конкретного компьютера, всё же можно указать какие-то общие черты, присущие большинству современных ЭВМ. Сейчас будет рассмотрено, что обычно входит в аппаратную реакцию процессора на сигнал прерывания.

Сначала надо сказать, что процессор "смотрит", пришел ли сигнал прерывания, только после выполнения очередной команды, таким образом, этот сигнал ждёт *завершения* текущей команды.¹ Здесь, однако, надо упомянуть о "нетипичной" команде в архитектуре рассматриваемого компьютера, это привилегированная команда с кодом операций **HLT** (HaLT). Эта команда останавливает выборку команд процессором, и только сигнал прерывания может вывести компьютер из этого "ничегонеделания". Обычно она используется в служебной программе для "выключения" процессора, если ни одна из программ не готова к счёту, это позволяет, в частности, снизить энергопотребление, что может оказаться важным, например, при работе портативного компьютера от аккумуляторов.

К описанному выше правилу начала аппаратной реакции на сигнал прерывания по завершению выполнения текущей команды необходимо сделать существенное замечание. Дело в том, что большинство современных ЭВМ отступают от принципа фон Неймана *последовательного* выполнения команд. Напоминаем, что согласно одному из положений этого принципа, следующая команда начала выполняться только после полного завершения текущей команды.

Современные ЭВМ могут одновременно выполнять несколько команд одного или даже *разных* потоков. Примерами компьютеров, обладающих такими возможностями, являются так называемые *конвейерные* ЭВМ, они могут одновременно выполнять несколько десятков команд. Для конвейерных ЭВМ необходимо уточнить, когда начинается аппаратная реакция на сигнал прерывания. Обычно это происходит после полного завершения *любой* из выполняющихся в данный момент команд. Выполнение остальных команд прекращается и в дальнейшем их необходимо будет повторить *сначала*, что обеспечивается специальным механизмом "отката" (retirement phase) программы на несколько команд назад (это занимает 20-30 тактов процессора).

Поэтому понятно, что конвейерные ЭВМ весьма "болезненно" относятся к сигналам прерывания, так как при этом приходится *заново повторять* последние, частично выполненные, команды прерванной задачи. Частый приход сигналов прерываний может сильно снизить скорость выполнения задач. Более подробно о конвейерных ЭВМ будет говориться в другой главе этой книги.

Сигналы прерываний приходят на специальную электронную схему ЦП, обычно она называется (программируемый) *контроллер прерываний* APIC (Advanced Programmable Interrupt Controller).² Именно у этого контроллера процессор "спрашивает", есть ли сигнал прерывания.

В многоядерных ЭВМ обычно у каждого процессорного ядра есть свой контроллер *внутренних* прерываний (ведь у каждого ядра они свои), и один на все ядра контроллер *внешних* прерываний (APIC IO). При этом каждое процессорное ядро после выполнения очередной своей команды "смотрит" на свой контроллер внутренних прерываний, а контроллер внешних прерываний сам посылает сигнал прерывания одному из внутренних контроллеров. Обычно сигнал прерывания с такого общего

¹ Даже если это деление на ноль или аппаратный сбой, всё равно можно считать, что центральный процессор *закончил* выполнение этой команды, хотя результат при этом, конечно, не определен. Отметим также, что перед командой могут располагаться вспомогательные команды-префиксы, которые не считаются "настоящими" командами, после их выполнения сигнал прерывания не анализируется, исключения составляют префиксы цикла **rep**, **repe** и **repne**, после выполнения этих команд-префиксов прерывания допускаются.

² Первая отечественная ЭВМ М40 с (очень примитивной) системой прерываний появилась в 1958 году. Первая PIC Intel 8229 для *персональных* ЭВМ появилась в 1978 году.

внешнего контроллера приходит к внутреннему контроллеру ядра, которое работает с наименьшим приоритетом.¹ При подаче сигнала прерывания некоторому ядру, контроллер повышает его приоритет до уровня приоритета устройства, подавшего сигнал прерывания, так что следующее внешнее прерывание будет, скорее всего, направлено уже другому ядру¹ [см. сноску в конце главы].

Этот механизм позволяет равномерно распределять нагрузку по обработке внешних прерываний между процессорными ядрами. Необходимо также отметить, что одно ядро, само выступая как внешнее устройство, может, используя APIC, отправить сигнал прерывания другому ядру, всем ядрам сразу и даже самому себе 😊.

Итак, если есть сигнал прерывания, то после окончания текущей команды процессор вырабатывает сигнал подтверждения приёма прерывания (Interrupt Acknowledge), после чего читает из контроллера номер сигнала прерывания (для нашего компьютера это целое беззнаковое число формата i8). Для некоторых из этих номеров пришедший сигнал прерывания *игнорируется*, и процессор продолжает выполнение команд текущего потока. Говорят, что прерывания с такими номерами в данный момент времени запрещены или *замаскированы*.² Для компьютера нашей архитектуры можно замаскировать все прерывания от *внешних* устройств (кроме прерывания с №2), установив в ноль значение специального *флага прерывания* с именем IF в регистре флагов EFLAGS. Кроме того, можно замаскировать каждое прерывание от внешних устройств по отдельности, для этого надо установить в ноль соответствующий этому прерыванию бит в *регистре маски* прерываний IMR контроллера. В принципе, и прерывание №2 тоже можно закрыть, но не с помощью флага IF, а с использованием нужного *порта* контроллера прерываний, о работе с портами будет рассказано в другой главе.

Маскировать сигналы прерываний о событиях в центральной части компьютера (например, деление на ноль, плохой код операции, сбой оперативной памяти и т.д.) нельзя, так как продолжение выполнения текущего потока без реакции на эти события не имеет смысла. Таким образом, прерывания с определёнными номерами можно закрывать (маскировать) и открывать (разрешать, снимать с них маску).

В том случае, если прерывание игнорируется (замаскировано), сигнал прерывания, тем не менее, продолжает оставаться на входе соответствующей схемы процессора до тех пор, пока маскирование этого сигнала не будет снято, или же на этот вход процессора ни придёт следующий сигнал прерывания. В последнем случае первый сигнал обычно безвозвратно потеряется, что может быть очень опасно, так как компьютер не прореагировал должным образом на некоторое событие, и оно прошло для него незамеченным. Чтобы снизить такую опасность, контроллер прерываний обычно запоминает приходящие сигналы в небольшой аппаратно реализованной кольцевой *очереди* для их дальнейшей обработки. Следовательно, для надёжной работы маскировать сигналы прерываний от внешних устройств можно только на некоторое весьма короткое время, после чего необходимо *открыть* возможность реакции на такие прерывания.

Теперь следует понять, зачем вообще может понадобиться маскировать некоторые прерывания. Дело в том, что, если пришедший сигнал внешнего прерывания не замаскирован, то, как вскоре станет ясно, процессор прекращает выполнение *текущего* потока и переключается на выполнение процедуры-обработчика данного прерывания (Interrupt Service Routine). Это может быть нежелательно и даже опасно, если текущий поток был занят срочной работой, которую нельзя прерывать даже на короткое время. Например, этот поток может обрабатывать некоторое важное событие, или же управлять каким-либо быстрым процессом во внешнем устройстве (линия связи с космическим аппаратом, химический реактор и т.д.).

С другой стороны, необходимо понять, что должны существовать и специальные *немаскируемые* сигналы прерывания от внешних устройств (Nonmaskable External Interrupts). В качестве при-

¹ Не надо путать (аппаратный) приоритет ядра IRQ (Interrupt ReQuest Level) с приоритетом выполняемого на этом ядре в данный момент вычислительного потока, приоритет потока назначается операционной системой.

² Такой термин появился потому, что разрешение прерывания обычно задаётся установкой в единицу определённого бита в некотором служебном регистре. На первых внешних носителях данных двоичная единица обозначалась отверстием (прорезью) в нужном месте картонной перфокарты, а ноль — отсутствием такого отверстия. Таким образом, наличие отверстия как бы позволяет "увидеть" пришедший сигнал прерывания, а иначе он становится невидимым, т.е. *замаскированным* (на него установлена маска).

меров таких событий можно привести нажатие кнопки Reset или выключение электрического питания стационарного компьютера. Надо сказать, что в этом случае компьютер останавливается не мгновенно, какую-то долю секунды он ещё может работать за счет энергии, накопленной на электрических конденсаторах в блоке питания. Этого времени хватит на выполнение нескольких миллионов команд и можно принять важные решения: послать сигнал тревоги, спасти ценные данные в энерго-независимой (Nonvolatile) памяти, переключиться на резервный блок питания и т.д. Как уже говорилось, нельзя маскировать внутренние сигналы прерываний, т.к. после такого сигнала продолжать выполнение текущего потока бессмысленно (деление на ноль, плохой код операции, отказ памяти и т.д.). Кроме того, как скоро станет ясно, внутренние прерывания возникают и при выполнении некоторых специальных команд, основным назначением этих команд как раз и является выдача сигнала прерывания (вскоре эти команды будут изучены). Для данной архитектуры все сигналы о событиях в центральной части компьютера замаскировать нельзя, и, как уже упоминалось, существует только одно немаскируемое прерывание от *внешних устройств* №2.

При описании архитектуры процессоров x86 внутренние прерывания называются *исключениями* (exceptions – имеется ввиду исключительная ситуация, возникшая при выполнении команды), а внешние – собственно прерываниями (interrupts). К внутренним прерываниям относят и так называемые *программные* прерывания, вызванные выполнением особых команд, основным назначением которых и является выработка сигнала прерывания. Исключения, как уже говорилось, часто называют *синхронными* прерываниями, т.к. они всегда возникают после выполнения вызвавшей их команды, а внешние прерывания возникают в любой момент, независимо (*асинхронно*) от выполнения команд.

В старших моделях семейства x86 все прерывания делятся на *нарушения* (fault), *ловушки* (trap) и *аварии* (abort).

- Нарушение (ошибка, отказ) можно определить до фактического завершения выполнения команды, поэтому счётчик адреса указывает не на следующую команду (как обычно), а на *текущую*, которая вызвала это нарушение, при этом в EFLAGS флаг RF (Restart Flag) повторения (рестарта) команды установлен в 1. Таким образом, эту команду можно при необходимости *повторить* (если, конечно, ошибку удалось исправить). В качестве примеров можно привести деление на ноль, чтение данных из отсутствующей в памяти страницы (после считывания её в память процедурой обработки этого прерывания, команду можно повторить). Кроме того, RF = 1, если нарушение произошло во время выполнения *строковых* команд с префиксом повторения (например, **rep movsb**), они не выполнились до конца и их надо *продолжить* с прерванного места.
- Ловушкой называется исключение, при котором текущая команда полностью выполнена и счётчик адреса указывает, как обычно, на *следующую* команду и можно *продолжить* выполнение прерванного потока. Так, например, работает отладчик при останове в контрольных точках или выполнении задачи в покомандном режиме. К ловушкам относится также большинство внешних прерываний (нажата кнопка клавиатуры, сдвинута мышка и т.д.). В этом случае текущая команда правильно выполнена до конца и (после обработки события) можно продолжить счёт со следующей команды.
- Авария свидетельствует о серьёзной ошибке в работе ЭВМ. При аварии счётчик адреса *не определён* (это, например, отказ аппаратуры, ошибки в системных таблицах, ошибках в шинах передачи данных или так называемая "двойная ошибка", когда процедура-обработчик сама вызывает исключение). При аварии продолжение выполнения прерванной задачи невозможно, она должна принудительно завершиться операционной системой.

Продолжим теперь рассмотрение аппаратной реакции на *незамаскированное* прерывание. Такая реакция сильно зависит от *режима* работы компьютера. Для рассматриваемой архитектуры это так называемый *реальный* режим (Real Mode), режим *виртуальной машины* MS-DOS (V86 Mode), *защищённый* режим (Protected Mode) и режим системного управления (System Management Mode). Будет рассмотрена реакция на сигнал прерывания только в защищённом режиме, именно здесь реализована плоская модель памяти, в которой работают приводимые в данной книге примеры программ на Ассемблере.

С точки зрения процессора, все сигналы прерываний можно разделить на "обычные" и "приоритетные" (далее будет сказано, как они различаются). Процедура-обработчик обычного прерывания

имеет тот же уровень привилегий, что и прерванный поток и может разделять его ресурсы, в частности, использовать его стек. В качестве примера такой процедуры-обработчика можно привести отладчик программ (как на языках высокого уровня, так и Ассемблера).¹ Отладчик расставляет в программе пользователя команды-ловушки (например, в так называемых контрольных точках), при выполнении таких команд возникают сигналы прерывания, обработкой которых и занимается отладчик.

Процедура-обработчик приоритетного прерывания должна для своей работы иметь более высокий уровень привилегий, чем прерванный поток пользователя. В связи с этим она располагается в *другом* сегменте кода и обязана иметь свой *собственный* (системный) стек, т.к., как уже говорилось, контроль привилегий требует, чтобы сегмент стека имел тот же уровень привилегий, что и сегмент кода. Процедура-обработчик приоритетного прерывания имеет возможность выполнять все команды машины, в том числе и привилегированные, запрещённые для обычных программ.

Каждый номер прерывания определяет для процессора так называемый *дескриптор* процедуры-обработчика прерывания с данным номером (по существу, это дескрипторы кодовых сегментов этих процедур). Все такие дескрипторы (каждый длиной по 8 байт) располагаются в специальной таблице дескрипторов прерываний IDT (Interrupt Descriptor Table), максимальная длина этой таблицы $8 \times 256 = 2048$ байт. По существу, номер прерывания является индексом в массиве IDT. На начало IDT указывает системный регистр IDTR, так что эта таблица может располагаться в любом месте физической памяти, а не обязательно с нулевого адреса, как было в младших моделях первого и второго поколений семейства x86. Для многоядерных ЭВМ у каждого процессорного ядра свой регистр IDTR и, соответственно, своя IDT.

При описании архитектуры рассматриваемого семейства дескрипторы процедур-обработчиков прерываний принято называть *шлюзами* (Gate), они "ведут" к процедурам-обработчикам прерывания. Естественно, различают шлюзы внешних прерываний (Interrupt Gate), ловушек (Trap Gate) и переключений задач (Task Gate). В каждом шлюзе (и, вообще, дескрипторе сегмента) есть поле с именем DPL (Descriptor Privilege Level), задающее уровень привилегий данного сегмента, если этот уровень выше привилегий текущего потока ($CPL > DPL$), то прерывание считается приоритетным. Приоритетные прерывания обслуживают шлюзы переключений задач (Task Gate), у них $DPL = 0$, такое прерывание будет *неприоритетным*, только если и $CPL = 0$ (в этом случае прерывание случилось, когда текущая задача сама работала в привилегированном режиме).

В начале аппаратной реакции на прерывание процессор автоматически запоминает в текущем стеке (для обычного прерывания) или отдельном системном стеке (для привилегированного прерывания) самую необходимую (минимальную) информацию о прерванном потоке. В некоторых книгах по архитектуре ЭВМ это называется *малым упрятыванием*, что хорошо отражает смысл такого действия.

На рис. 7.1 показаны стек задачи и системный стек (после малого упрятывания) для обычного и приоритетного прерывания соответственно. Индекс Р у регистра обозначает регистр прерванного потока, а индекс I – регистр процедуры-обработчика прерывания. Отметим, что для приоритетного прерывания используется только системный стек, а стек прерванной программы не меняется.

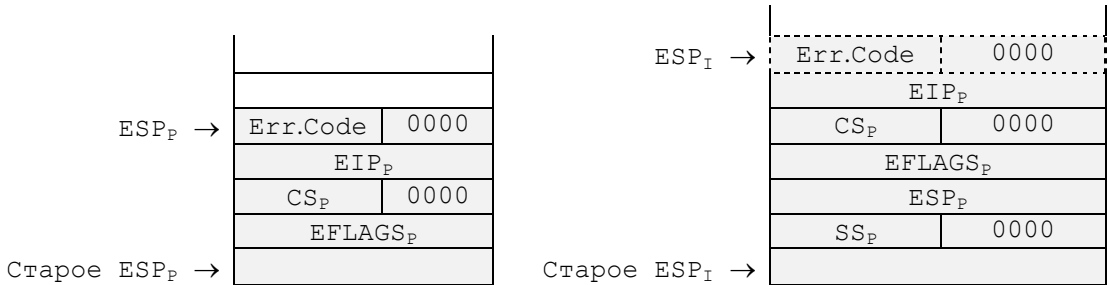


Рис. 7.1. Вид стека программы и системного стека после малого упрятывания.

Как видно, для текущего потока в стек записывается регистр флагов EFLAGS, регистры EIP и CS для обеспечения возможности возврата, а для некоторых исключений также (необязательный) код

¹ Здесь надо отметить, что, наряду с отладчиками прикладного уровня (например OllyDbg, в просторечии "Ольга"), которые работают на прикладном уровне, существуют и отладчики, работающие на уровне ядра (например, IDA Pro, в просторечии "Ида"), их вызов может производиться и приоритетным прерыванием.

ошибки (Error Code), 16-битные сегментные регистры расширяются до 32-х бит. Код ошибки содержит некоторую дополнительную информацию, специфичную для конкретного прерывания, обычно там селектор "плохого" дескриптора, при использовании которого возникло прерывание. Для приоритетного прерывания в системный стек предварительно записываются регистры SS и ESP прерванной задачи, что даёт обработчику прерывания возможность анализировать этот "чужой" стек.

Как видим, эти действия похожи на действия при выполнении команды дальнего вызова процедуры через шлюз вызова (Call Gate) по команде **call**. Да и назначение у них одно – обеспечить возможность возврата в прерванное место текущей программы. Из этого следует, что стек должен быть у любой программы, даже если она сама им и не пользуется.ⁱⁱ [см. сноску в конце главы]

Далее действия процессора существенно различаются для привилегированного и непривилегированного прерывания. Для привилегированного прерывания работает так называемый механизм переключения задач (напомним, что в этом случае обработчик прерываний работает в привилегированном режиме и в своём адресном пространстве, в частности, со своим стеком). Переключение на новую задачу производится не только по сигналам приоритетного прерывания, но, как уже говорилось ранее, также по командам **call far ptr** и **jmp far ptr**, когда в качестве поля `seg` задан селектор TSS.

При переключении задач процессор для прерванного потока автоматически запоминает информацию, необходимую для возобновления его счета. Необходимо запомнить все сегментные, адресуемые, а также некоторые управляющие и системные регистры. Вся эта информация запоминается в специальной области памяти, связанной с прерванным потоком, эта область памяти называется в литературе по-разному: информационным полем, полем сохранения, контекстом процесса, а в архитектуре процессоров x86 *контекстом задачи*.

Контексты всех задач хранятся в хорошо защищенном месте памяти (обычно в ядре операционной системы), в старших моделях семейства x86 контекст задачи хранится в области памяти, которая называется сегментом состояния задачи TSS (Task State Segment).¹ Каждый TSS определяется своим дескриптором, все такие дескрипторы хранятся в глобальной дескрипторной таблице GDT. Селектор (номер) TSS в GDT хранится в системном регистре состояния (выполняемого) потока TR (Task Register). Регистр TR, свой для каждого процессорного ядра, является "самым главным" регистром, он хранит селектор дескриптора TSS текущего потока, а в теневой части этого регистра содержится и сам дескриптор. Итак по цепочке

TR (селектор TSS в GDT) → дескриптор TSS из GDT → адрес TSS

процессор получает адрес контекста прерванного потока и спасает туда всю информацию нужную для возобновления его счёта с прерванного места. В частности, это значение управляющего регистра CR3, где хранится ссылка на каталог всех "личных" страниц задачи в виртуальной памяти, и системный регистр LDTR с селектором локальной дескрипторной таблицы DTR. Таким образом, в TSS автоматически сохраняется практически *полная* информация о задаче. Кроме того, у TSS есть как обязательная часть, с которой автоматически работает процессор (её длина 104 байта), так и дополнительная часть, размер которой устанавливает программист. В дополнительной части можно хранить любую информацию, описывающую поток.

Далее процессор извлекает из дескриптора-шлюза задачи IDT селектор TSS обработчика прерывания, записывает его в регистр задач TR (в теневую часть этого регистра загружается и сам дескриптор задачи), после чего загружает в процессор из TSS обработчика контекст новой задачи (фактически это и есть *переключение* на новую задачу). Когда такое переключение производится по прерыванию или по команде дальнего вызова процедуры **call far ptr**, то в поле обратной связи PTL (Previous Task Link) в TSS новой задачи сохраняется селектор TSS *предыдущей* задачи, с которой

¹ Для продвинутых читателей. TSS считается *специальным* сегментом памяти, это задаётся битом S=0 (системный объект) в дескрипторе TSS. Как следствие, селектор TSS нельзя загрузить в сегментный регистр (только в TR), и никакая программа (даже с нулевым уровнем привилегий), не может читать и писать в этот сегмент, доступ к нему имеет только сам процессор. Поле DPL дескриптора TSS означает не уровень привилегий этого сегмента (так как к нему не может обратиться ни одна программа), а только уровень привилегий, требуемых для обращения к задаче с данным TSS. При создании *нового* TSS операционная система создаёт *псевдоним* сегмента (как сегмент данных), чтобы заполнить TSS нужными значениями, затем этот псевдоним уничтожается. Всё это показывает *степень защиты* контекстов задач.

произошло переключение. Это позволяет, по аналогии с вызовом процедуры, произвести из текущей задачи возврат в предыдущую, выполнив команду **iret**, и возобновить счёт прерванного потока. Поле PTL не заполняется и, соответственно, прямой возврат по **iret** невозможен, при безусловном переключении на новую задачу по команде **jmp far ptr**. К сожалению, рекурсивный вызов задач запрещён (вызывает прерывание). Необходимо учитывать, что переключение на новую задачу занимает в 20-30 раз больше времени, чем дальний (межсегментный) вызов процедуры (около 1000 тактов ЦП). На рис. 7.2 показана схема переключения задач для приоритетного сигнала прерывания.ⁱⁱⁱ [см. сноску в конце главы]

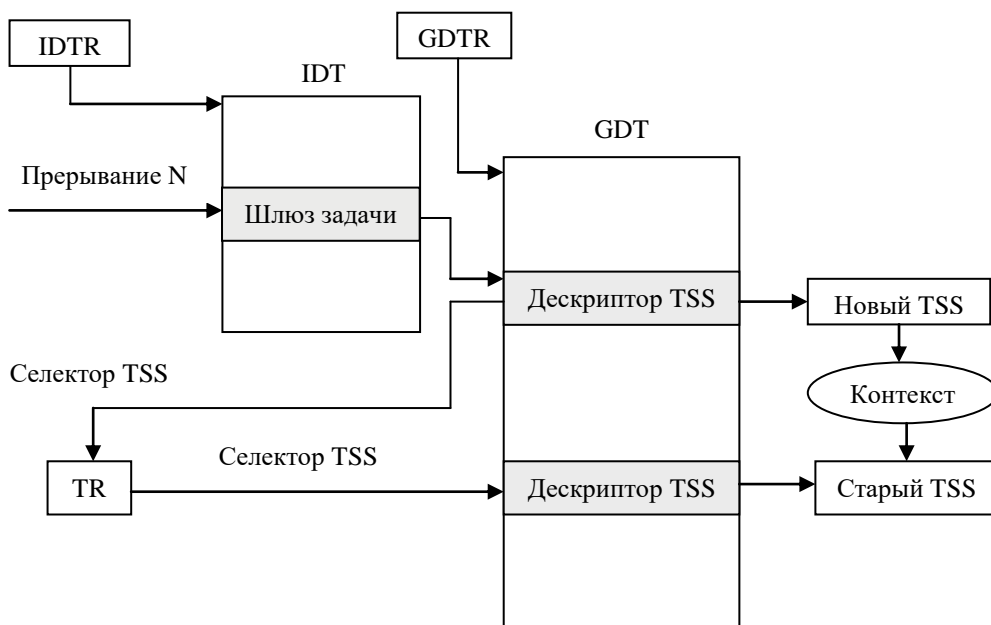


Рис. 7.2. Схема переключения задач для приоритетного сигнала прерывания (TR – Регистр (текущей) задачи, IDT – Таблица дескрипторов прерываний, GDT – Глобальная таблица дескрипторов, TSS – Сегмент состояния задачи).

Вот теперь аппаратная реакция на привилегированное прерывание закончилась и начинает выполняться (привилегированная) процедура-обработчик данного прерывания.¹

Для непривилегированного прерывания действия процессора много проще, так как считается, что процедура-обработчик прерывания работает в рамках текущего потока, и переключения контекстов не требуется. Поэтому, после сохранения в текущем стеке минимальной информации, как показано на рис. 7.1, процессор извлекает из дескриптора-шлюза и присваивает регистрам CS и EIP новые значения, которые являются дальним адресом оперативной памяти, где находится процедура-обработчик данного прерывания. Таким образом, у процедуры-обработчика прерывания может быть свой сегмент кода, но сегмент данных и стека она наследует от прерванного потока. Заметим, что наличие собственного сегмента кода гарантирует, что прерванный поток не мог "испортить" процедуру-обработчика прерывания. Современные ОС практически не используют непривилегированные прерывания.

Непосредственно перед переходом на процедуру-обработчика процессор автоматически выполняет ещё некоторые действия. Для нового потока сбрасываются (очищаются) определённые биты в служебных регистрах, в частности, в регистре EFLAGS это флаги TF и IF (флаг IF не меняется только при вызове процедуры-обработчика через шлюз ловушки).

Таким образом, чаще всего процедура-обработчик *начинает* своё выполнение в режиме запрета внешних прерываний. Это гарантирует, что, начав свою работу, процедура-обработчик не будет тут же (после выполнения первой же команды) прервана другим внешним сигналом прерывания. Как уже

¹ Точнее, порождается новый поток, который называется отложенной процедурой обработки прерывания, этот поток ставится в очередь на выполнение в диспетчер задач Windows. Такой поток имеет приоритет (Dispatch Level), который ниже, чем у процедур обработки прерываний, но выше, чем приоритет прикладных задач пользователя (Passive Level), и, следовательно будет выполняться перед программами пользователей.

говорилось, значение флага `IF=0` маскирует все прерывания от внешних устройств, кроме прерывания с №2. Флаг `TF:=0` устанавливается потому, что при значении `TF=1` процессор всегда посылает *сам себе* (немаскируемый) сигнал об исключении `EXCEPTION_SINGLE_STEP` с номером `N=1` после выполнения *каждой* команды.¹ Этот флаг используется при работе программ-отладчиков для пошагового выполнения (трассировки) отлаживаемой программы. Надеюсь, что Вы уже имели возможность познакомиться с работой отладчика при программировании на языке высокого уровня. На этом *аппаратная* реакция на незамаскированное прерывание заканчивается.

Заметим, что некоторым аналогом аппаратной реакции ЭВМ на прерывание в живой природе является *безусловный рефлекс*. Безусловный рефлекс позволяет живому существу "автоматически" (а, следовательно, быстро, "не раздумывая") реагировать на произошедшее событие. Например, если человек обжигает пальцы на огне, то сначала он автоматически отдергивает руку, а лишь потом начинает разбираться, что же собственно произошло. Так и компьютер по сигналу прерывания может автоматически, "не раздумывая" переключиться на процедуру-обработчика этого сигнала. Отсюда можно сделать вывод, что одно из назначений системы прерываний – обеспечить быструю реакцию компьютера на события, возникающие как в его центральной части, так и на периферии (в устройствах ввода/вывода).

Итак, после завершения аппаратной реакции на незамаскированный сигнал прерывания начинает работать процедура-обработчик прерывания с данным номером, эта процедура выполняет, как говорят, *программную* реакцию на прерывание. Заметим, что обработчик прерывания *не является* в полном смысле процедурой, как в языках высокого уровня или в Ассемблере, так как вызывается не командой `call`. Некоторая аналогия, однако, сохраняется, так как поле `PTL` в `TSS` позволяет возвращаться в предыдущий (прерванный) программный поток по команде `iret`.

Аналогом программной реакции на прерывание в живой природе можно считать условный рефлекс. Как у живого организма можно выработать условный рефлекс на некоторый внешний раздражитель (сигнал), так и компьютер можно "научить", как реагировать на то или иное событие, написав процедуру-обработчик сигнала прерывания с номером, соответствующим этому событию.

Из рассмотренной выше схемы аппаратной реакции на сигнал прерывания можно сделать вывод о том, что наш компьютер будет правильно работать только тогда, когда его таблица дескрипторов прерываний заполнена не случайными числами, а "правильными" данными процедур-обработчиков прерываний. Об этом должна позаботиться операционная система нашего компьютера в самом начале своей работы.

Рассмотрим теперь схему работы процедуры-обработчика прерывания. Напомним, что эта процедура чаще всего начинается выполняться в режиме с *закрытыми* прерываниями от внешних устройств. Как уже говорилось выше, долго работать в таком режиме очень опасно, и следует как можно скорее разрешить (открыть) такие прерывания. Для нашего компьютера это установка в единицу флага `IF` в регистре `EFLAGS`, что можно сделать, выполнив привилегированную команду ввода/вывода `sti`.

Действия, выполняемые в режиме с закрытыми прерываниями, обычно называются *минимальной программной реакцией* на прерывание. Для обработчика неприоритетного прерывания, как и для обычной процедуры, необходимо запомнить (в стеке) все регистры, которые предполагается менять. Для обработчика приоритетного прерывания этого, естественно, делать не надо, эта информация уже спасена в `TSS`. Затем контроллеру прерывания надо сообщить о том, что прерывание выбрано на обработку (послать, выполнив специальную команду, так называемый *конец прерывания* `EOI`), это удаляет сигнал прерывания из очереди контроллера.

¹ Кроме некоторых "хитрых" команд, прерывание после выполнения которых никогда не происходит. Например, это команды, которые загружают сегментный регистр `SS`, в частности, нет прерывания после команды `mov ss, new_ss`, так как следом скорее всего должна идти команда `mov esp, new_esp`, и при прерывании между ними система рухнет (нет нормального стека). Вообще-то есть команда `lss esp, m48`, *одновременно* загружающая регистры `SS` и `ESP`, но сейчас мало программистов о ней знают, а в 64-битной версии таких команд уже нет (значения сегментных регистров `ds`, `es` и `ss` всегда тождественны). Кроме того, современные процессоры позволяют, установив в управляющем регистре флаг `BTS` (Branch Trace Store), генерировать прерывания по флагу `TF` только после команд перехода или при возникновении исключений.

Далее выполняются самые необходимые действия, связанные с произошедшим событием. Например, если нажата или отпущена клавиша на клавиатуре, то это надо где-то зафиксировать (например, запомнить в очереди введенных с клавиатуры событий). Если этого не сделать на этапе минимальной реакции и открыть прерывания, то процедура-обработчик может быть надолго прервана новым сигналом прерывания. Этот сигнал произведет переключение на какую-то другую процедуру-обработчик, за время её работы уже может быть нажата новая клавиша, а информация о нажатой ранее клавише, таким образом, может быть безвозвратно потеряна.

После выполнения минимальной программной реакции процедура-обработчик включает (разрешает) внешние прерывания – в нашей архитектуре устанавливает флаг командой **sti** (SeT Interrupt, $IF:=1$). Далее производится *полная программная реакция* на прерывания, т.е. процедура-обработчик выполняет *все* необходимые действия, связанные с произошедшим событием. Обычно для выполнения полной программной реакции порождается новый поток, так называемая отложенная процедура **DPC** (Deferred Procedure Call), её приоритет выше, чем у обычных прикладных задач, но ниже, чем у процедур, выполняющих минимальную программную реакцию.

Вообще говоря, допускается, что выполнение такой отложенной процедуры может быть прервано другим внешним сигналом прерывания. В частности, это может быть и сигнал с таким же номером **N**, при этом произойдёт *повторный* вход в начало этой же самой процедуры-обработчика. Те программы, которые допускают такой "принудительный" повторный вход в своё начало (не путать это с рекурсивным вызовом!), называются повторно-входимыми или "по-иностранному" *реентерабельными* (reentrant). Для реентерабельных программ при каждом новом входе в их начало порождается *новый поток* и должна резервироваться новая область памяти под хранение контекста задачи при её прерывании, т.е. создаётся новый дескриптор **TSS** в таблице **GDT**. Программы обработки прерываний для младших моделей семейства часто не были реентерабельными, что порождало определённые проблемы, которые здесь обсуждаться не будут.

Необходимо также понять, что при работе процедуры-обработчика выполнении некоторых участков её программы нельзя прерывать, иначе может возникнуть ошибка. В программировании такие участки программы называются критическими секциями (critical section). Надо позаботиться о том, чтобы при выполнении каждой критической секции работа не была прервана. Для этого в начале каждой такой секции стоит команда запрета прерывания от внешних устройств **cli**, а в конце секции – команда открытия таких прерываний **sti**. Таким образом, режим запрета прерывания от внешних устройств может многократно включаться и выключаться. Разумеется, внутри критических секций не должно быть команд, которые сами могут вызвать прерывания. При работе в защищенном режиме **Windows** пользовательская программа и процедуры-обработчики неприоритетных прерываний не могут сами закрывать и открывать внешние прерывания этими командами. Они вынуждены использовать предназначенные для этого системные вызовы **EnterCriticalSection** и **LeaveCriticalSection**. Здесь необходимо также учитывать, что в многоядерных ЭВМ простого запрета прерываний от внешних устройств для защиты критической секции недостаточно, так как в это же место программы может зайти и поток, выполняющийся на другом ядре (там свой регистр **EFLAGS** со своим флагом **IF**). Упомянутые выше системные вызовы обеспечивают такую, более сложную защиту критических секций.¹

Во время своей работы процедура-обработчик прерывание удаляет из стека и код ошибки (**Error Code**), если он там был. Закончив полную обработку сигнала прерывания, процедура-обработчик должна обеспечить возобновление счёта потока, прерванного сигналом прерывания.

Возврат из непривилегированного прерывания делается почти так же, как и возврат из процедуры. Сначала, как обычно, стек очищается от локальных переменных и из него восстанавливаются значения "испорченных" регистров. Затем выполняется команда без параметров **iret**. В отличие от обычной команды возврата из процедуры **ret**, эта новая команда извлекает из стека и восстанавливает старое значение не одного регистра **EIP**, а трёх регистров **EIP**, **CS** и **EFLAGS** (см. рис. 7.1). Так как у команды **iret** нет параметра, то и дополнительная очистка стека, как в команде **ret n**, не производится.

¹ Критические секции можно использовать только для синхронизации потоков внутри одной задачи, для синхронизации независимых задач используются другие средства (семафоры, мьютексы и т.д.), эта тема изучается в курсе по ОС.

При возврате из привилегированного прерывания различают два случая. Когда переключение на новую задачу произведено с возможностью возврата (с заполнением поля PTL селектором TSS прерывающего потока), тогда всё просто.¹ В этом случае команда **iret** сама производит (обратное) полное переключение на контекст прерванного потока и возобновляется его выполнение.

Рассмотрим теперь случай, когда возобновление счёта прерванного потока по команде **iret** невозможно. Во-первых, это случай переключения на новую задачу без заполнения поля возврата PTL в TSS (например, по команде **jmp far ptr**). Во-вторых, часто сам обработчик прерывания не принимает решение, что необходимо *немедленное* возобновление счёта прерванного потока. Например, прерванный поток может ждать наступление некоторого события, скажем, сигнала об окончании чтения с диска нужной ему порции данных. Такие потоки после обработки прерывания переводятся в состояния ожидания, решение о возобновлении их счёта будет принимать системная программа – *диспетчер задач* (Task Manager).

Диспетчер управляет счётом всех потоков, так как имеет список всех селекторов задач из GDT. Те задачи, у которых нет поля PTL в TSS, называются в диспетчере Windows *приложениями*, они могут порождать подчинённые задачи (называемые в диспетчере *процессами*). Для каждой задачи диспетчер в отдельной колонке показывает число порождённых ей потоков. При уничтожении приложения уничтожаются и все его подчинённые задачи, можно, однако, выборочно уничтожать и отдельную задачу, при этом уничтожаются все её потоки (уничтожение отдельного потока, однако, в диспетчере задач не предусмотрено). Диспетчер может приостановить и возобновить счёт задачи. Каждой задаче присвоен числовой приоритет, и, если прерванных и готовых к продолжению своего счёта задач больше одной, то возобновляется выполнение задачи с наибольшим приоритетом. Полностью эта тема изучается в курсе по операционным системам.

Из рассмотренной выше схемы механизма обработки прерываний можно сделать один важный вывод. Для "обычной" программы прерывание проходит *незамеченным*, если только эта программа специально не следит за показанием встроенных в компьютер часов (в последнем случае она может заметить, что физическое время её счёта "почему-то" увеличилось). Иногда для подчеркивания этого факта говорят, что механизм прерываний *прозрачен* для выполняемой программы обычного пользователя. В Таблице 7.1 изображены некоторые позиции вектора дескрипторов прерываний.

Продолжим теперь изучение специальных *команд* прерываний, выполнение каждой такой команды *всегда* вызывает прерывание (исключением является команда **into**, которая вызывает прерывание *не всегда*, эта команда рассматривается далее). Таким образом, каждая команда прерывания тоже является командой перехода.

Сначала рассмотрим некоторые *команды* прерывания, о которых упоминается в Таблице 7.1. Команда **int** является самой короткой (длиной в один байт) командой, которая всегда вызывает прерывание с номером N=3. В основном эта команда используется при работе *отладчика* – специальной программы, облегчающей программисту разработку новых программ. Отладчик, в частности, может вставлять в отлаживаемый программный код так называемые *контрольные точки* (Break Point, на жаргоне программистов *бряк*) – это те места, в которых отлаживаемая программа должна передать управление отладчику. Для такой передачи хорошо подходит короткая команда **int**, если программа-отладчик реализована в виде обработчика прерывания с номером N=3 (так как эта команда самая короткая по длине, то её можно *временно* подставить на место *любой* машинной команды в отлаживаемой программе).²

Команда **into** реализует *условное* прерывание: при выполнении этой команды прерывание происходит только в том случае, если флаг OF=1, иначе продолжается последовательное выполнение программы. Основное назначение команды **into** – *эффективно* реализовать надежное программирование при обработке целых *знаковых* чисел. Как Вы уже знаете, при выполнении операций сложения и вычитания со знаковыми числами возможна ошибка, при этом флаг переполнения OF устанавливается в единицу, но выполнение программы продолжается. Кроме того, вспомним, что флаг пере-

¹ В этом случае в регистре EFLAGS установлен в единицу флаг NT (Nested Task – вложенная задача).

² В современных процессорах есть специальные отладочные регистры, адреса в которых задают контрольные точки и в командах, и в данных, если этих регистров хватает (а в нашей архитектуре их всего 4: DR0–DR3), то команды **int** в программу можно не ставить.

полнения устанавливается в единицу и при операциях умножения, если *значащие* биты результата попадают в *старшую* часть произведения.

Таблица 7.1. Начало вектора дескрипторов прерываний

Номер	Описание события	Тип
N=0	Ошибка целочисленного деления (#DE – Divide Error)	Нарушение
N=1	Установлен флаг TF=1 (#DB – DeBug)	Ловушка
N=2	Немаскируемое внешнее прерывание (NMI)	Прерывание
N=3	Выполнена команда int (#BP – BreakPoint)	Ловушка
N=4	Команда into при OF=1 (#OF – OverFlow)	Ловушка
N=5	Команда bound определила выход индекса за границы массива (#BR – Bound Region)	Нарушение
N=6	Команда с плохим КОП (#UD – Invalid Opcode) ¹	Нарушение
N=8	Двойная ошибка (#DF – Double Fault), при обработке исключения возникло ещё одно исключение	Авария
N=9	Зарезервировано	
N=10	Плохой TSS	Нарушение
N=11	Отсутствие сегмента в оперативной памяти	Нарушение
N=12	Переполнение стека	Нарушение
N=13	Ошибка общей защиты (#GP – General-Protection Exception)	Нарушение
N=14	Нарушение доступа к странице виртуальной памяти (#PF – Page Fault)	Нарушение
N=16	Ошибка при операциях с вещественными числами	Нарушение
N=17	Контроль выравнивания: магическое условие: ☺ CR0 [AM]=1 & EFLAGS [AC]=1 & CPL=3	Нарушение
N=18	Ошибка машинного контроля (#MC – Machine Check)	Авария

При надёжном программировании проверку флага переполнения необходимо ставить после каждой такой команды. Для такой проверки хорошо подходит команда **into**, так как это самая короткая (однобайтная) команда условного перехода по значению OF=1 (обычная команда **jc Error** при дальнейшем переходе занимает 5 байт). При этом, правда, обработку аварийной ситуации должна производить процедура-обработчик прерывания с номером N=4.²

Прерывание с номером N=4 может произойти при выполнении команды

bound op1, op2

Таблица допустимых операндов для этой команды:

op1	op2
r16	m16+m16
r32	m32+m32

Эта команда используется для проверки выхода адреса элемента массива за допустимые границы (check array BOUNDS). Первый операнд-регистр задаёт проверяемый регистр, а второй операнд задаёт расположенные в памяти нижнюю LB (Lower Bound) и верхнюю UB (Upper Bound) границы адреса памяти (два слова m16 или двойных слова m32 соответственно, LB<=UB). Например:

¹ Мнемоника #UD (UNdefined) намекает на особую команду без параметров с кодом операции **ud2** (код 0Fh 0Bh), эта несуществующая команда, она всегда вызывает нарушение #UD. Кроме того, это нарушение возникает, если код операции не соответствует операндам, например, **jmp far ptr eax** (дальний переход без параметра m48). Можно также отметить, что, кроме команд с плохими кодами операций, в архитектуре x86 встречаются и так называемые *недействительные* команды, выполнение которых не вызывает прерывания. Пользователи называют их *недокументированными* командами, фирма Intel не гарантирует, что они останутся в следующих версиях процессоров. В качестве примера можно привести команду без явных операндов с кодом 0D6h, в старой документации она имела мнемонику **s[et]alc** (SET to AL Cf), пока эта команда выполняется по правилу

if CF=0 then AL:=0 else AL:=0FFh

Отметим, что команда **setalc** работает так же, как и команда **sbb al, al**, но не меняет флагов. Современные Ассемблеры команду **setalc** "не понимают", но, если поставить в программу код операции этой команды в виде константы **db 0D6h**, то процессоры (пока?) спокойно выполняют такую команду.

² К сожалению, в 64-разрядном режиме эта команда не работает.

```

.data
Mas dd 1000 dup(?)
Bnd dd Mas,Bnd-4; Границы массива
.code
mov     eax,offset Mas
bound   eax,Bnd; Всё хорошо
mov     eax,offset Bnd; За границами
bound   eax,Bnd; Нарушение STATUS_ARRAY_BOUND_EXCEEDED 1

```

И, наконец, рассмотрим команду языка машины, которая всегда вызывает прерывание с номером N, заданным в качестве её операнда:

```
int op1
```

Здесь `op1=N` имеет формат `i8`. Напомним, что в рассматриваемой архитектуре такие прерывания называются *программными исключениями*. Заметим, что с помощью команды `int` можно вызвать прерывание с любым номером, например прерывание, соответствующее делению на ноль или плохому коду операции (правда, в этом случае процедуре-обработчику не передаётся код ошибки `Error Code`, о котором говорилось выше). Более того, прерывания с номерами, большими 31_{10} , в данной архитектуре можно вызвать, *только* выполняя команду `int N` с соответствующим параметром-номером прерывания, поэтому их часто так и называют – *программные прерывания*. Используя программные прерывания, легко отлаживать процедуры-обработчики прерываний, но основное назначение таких команд другое.

Дело в том, что в большинстве программ необходимо выполнять некоторые широко распространенные действия (обмен данными с внешними устройствами, выполнение определенных стандартных операций и многое другое). Естественно желание реализовать такие широко распространенные для большинства программ действия в виде некоторого общедоступного *набора процедур*, чтобы не реализовывать эти действия в каждой программе, а просто вызывать для этого необходимые стандартные процедуры. Обычно процедуры, реализующие эти действия, оформляются в виде библиотеки стандартных процедур и всегда находятся в оперативной памяти компьютера. Так как адреса этих процедур могут меняться (например, при их модификации), то лучше всего присвоить каждой такой процедуре свой *номер* N и оформлять такие процедуры в виде обработчиков прерываний с этим номером. Это освобождает остальные программы от необходимости знать конкретное месторасположение таких процедур в памяти, достаточно знать только их *номера*. В этом случае вызов конкретной процедуры с номером N следует производить командой `int N`.

Исходя из описанного выше, такие команды прерывания называют *системными вызовами* (системными функциями операционной системы), а соответствующую библиотеку стандартных процедур – Базовой системой процедур ввода/вывода (английское сокращение BIOS – Base Input/Output System). Параметры для таких процедур обычно передаются на регистрах, т.е. для системных вызовов BIOS не выполняются стандартные соглашения о связях.²

Современные операционные системы (в частности, Windows) запрещают программам обычных пользователей (работающих в непривилегированном режиме) использовать для системных вызовов BIOS машинные команды `int N`. Вместо этого программисту необходимо использовать библиотечные процедуры операционной системы. Например, в Windows эти процедуры собраны в библиотеку WinAPI (Windows Application Programming Interface), в ней много тысяч таких процедур. Сис-

¹ С 2012 года в 64-разрядном режиме работы процессоров Intel появилась технология расширения защиты памяти MPX (Memory Protection Extensions) где есть уже *четыре* 128-разрядных регистра защиты BND0–BND3, в каждом из которых можно задать LB (Lower Bound) и UB (Upper Bound) границы контролируемого доступа к памяти. Есть также регистр состояния BNDSTATUS и новые команды для работы с этими регистрами. В связи с этим команда `bound` больше не работает.

² Первые BIOS появились, когда ещё не существовало многоязыковых систем программирования, и не было необходимости в стандартных соглашениях о связях. Заметим также, что вызов с нестандартными соглашениями о связях (через регистры) производится быстрее. При этом, однако, многие вещи являются невозможными, например, рекурсивный вызов таких процедур. После 2007 года ЭВМ вместо BIOS снабжены UEFI (Unified Extensible Firmware Interface) – по сути это небольшие 64-разрядные операционные системы (с графическим интерфейсом и возможностью выхода в сеть), управляющие начальной загрузкой компьютера. Однако, чтобы не путать пользователей, они тоже часто называются BIOS.

темный вызов в API выполняется, как в языке высокого уровня, т.е. у системного вызова есть имя и список формальных параметров. Таким образом, исходные системные вызовы BIOS не используются, что, наверное, удобнее для многих программистов.

С языка Ассемблера остаётся возможность делать системные вызовы "вручную" командой Ассемблера `int 2Eh`,¹ её параметр задаёт селектор 2Eh, который является в IDT шлюзом задачи диспетчера системных вызовов. Номер системной процедуры передаётся на регистре EAX, а адрес списка параметров на регистре EDX. Процедура-обработчик прерывания 2Eh использует номер системной процедуры, заданный на регистре EAX, как индекс в таблице системных вызовов SDT (Service Descriptor Table), переключаясь на выполнение конкретного запроса.

На современных процессорах вместо системного вызова с помощью команды прерывания `int 2Eh` рекомендуется использовать новую команду быстрого системного вызова **sysenter**, (**syscall** в 64-битном режиме), она может быть выполнена с любого уровня привилегий.

Номер системной процедуры передаётся для **sysenter** на регистре EAX, первые пять параметров для системного вызова на регистрах (EBX, ECX, EDX, ESI и EDI), остальные в стеке. В начале своей работы сама системная процедура сначала заносит в стек (пользователя) адрес возврата (сама команда **sysenter** этого не делает!), затем регистры ECX, EDX и EBP записываются в стек (именно в таком порядке), после чего регистр ESP копируется в EBP. Таким образом, по аналогии с вызовом процедуры, в стеке пользователя создаётся стековый кадр, показанный на рис. 7.3. Смысл таких действий будет понятен при описании работы команды возврата **sysexit**.²

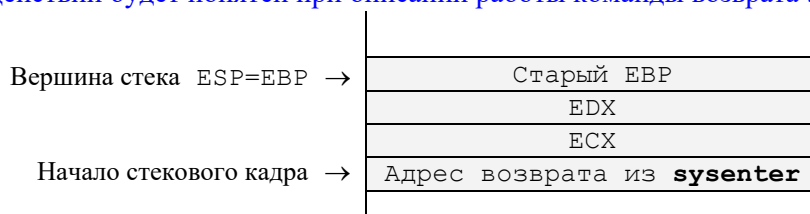


Рис. 7.3. Вид стекового кадра пользователя для команды

В отличие от вызова через прерывание `int 2Eh`, команда **sysenter** осуществляет системный вызов, не производя прерывания, нужная минимальная информация для запуска программы системного вызова извлекается не из таблиц дескрипторов, расположенных в памяти, а прямо со служебных, так называемых MSR (Model-Specific Register) регистров процессора:

```
CS:=IA32_SYSENTER_CS; Теневой регистр cs: начало=0, предел=0FFFFFFFFh
EIP:=IA32_SYSENTER_EIP; Точка входа в диспетчер системных вызовов
SS:=IA32_SYSENTER_CS+8; Теневой регистр ss: начало=0, предел=0FFFFFFFFh
ESP:=IA32_SYSENTER_ESP; Системный стек (а старый ESP в EBP)
```

По существу, это самое быстрое переключение на приоритетную процедуру системного вызова, так как при отсутствии прерывания не сбрасываются кэш и конвейер, а также нет переключения контекста. При входе по команде **sysenter** (**syscall**) для выхода из системного вызова используется команда **sysexit** (соответственно, **sysret** в 64-битном режиме), они могут использоваться только для возврата с уровня ядра на уровень пользователя. При выполнении команды **sysexit**:

```
CS:=IA32_SYSENTER_CS+16
EIP:=EDX
SS:=IA32_SYSENTER_CS+24
ESP:=ECX
```

¹ В ОС семейства UNIX для системного вызова используется команда Ассемблера `int 80h`.

² В 64-битном режиме используется команда **syscall** (соответственно **sysret**) и всё сильно упрощается. Номер системного вызова загружается в регистр RAX, а шесть первых параметров (если они есть) соответственно, в регистры RDI, RSI, RDX, R10, R8 и R9. Регистры RCX и R11 портятся, так как **syscall** присваивает (для последующего выполнения **sysret**) RCX:=RIP и R11:=RFLAGS, RSP не меняется. При возврате по команде **sysret**, наоборот RIP:=RCX и RFLAGS(почти весь):=R11, все регистры (кроме RCX и R11) сохраняются, возвращать результата на RAX.

Теперь понятно, для чего в стеке спасались регистры ECX и EDX, они портятся при возврате, так как перед выполнением команды **sysexit** процедура системного вызова должна выполнить следующие действия:

```
mov ecx,[ebp];    Старый ESP
mov edx,[ebp+16]; Старый EIP
sysexit
```

Ясно, что каждый системный вызов WinAPI ведёт в одну из служебных функций DLL-библиотеки (обычно ntdll.dll)¹, которая, в свою очередь, содержит внутри себя команду **sysenter**, ведущую в процедуру-обработчик прерывания (обычно в ядре операционной системы).

Ниже приведён пример прямого вызова служебных процедур ядра с помощью команды **sysenter**. Для вывода текстовой строки в нашем Ассемблере можно использовать как макрокоманды, так и напрямую обратиться к ядру ОС командой **sysenter**:

```
.data
    Mes db 'Hello, word!',0
    Len equ $-Mes; Длина выводимой строки
.code
; Вызов через макрокоманду:
    outstr offset Mes
; Вызов через sysenter:
    mov  eax,4;           № системного вызова sys_write
    mov  ebx,1;           1-й параметр=дескриптор stdout
    mov  ecx,offset Mes;  2-й параметр=адрес строки
    mov  edx,Len;         3-й параметр=длина строки
    push Cont;            точка возврата из sysenter
    push ecx
    push edx
    push ebp
    mov  ebp,esp;         ebp -> стековый кадр для sysenter
    sysenter
Cont:      ;              сюда возврат из sysenter
    pop  ebp
    pop  edx
    pop  ecx;             восстановление регистров2
; другие регистры (в частности, EFLAFS) при необходимости
; должен запомнить и восстановить сам системный вызов
```

Таким образом, команды **sysenter** и **sysexit** требуют, чтобы в таблице дескрипторов было специальное расположение дескрипторов сегментов кода и стека программы пользователя по отношению к дескрипторам сегментов кода и стека процедуры системного вызова (+8 и +16).

Для продвинутых читателей. Как видно, хотя вызов системной процедуры теперь не требует прерывания и переключения контекста, но всё ещё остаётся много стековых операций. Для дальнейшего ускорения системного вызова в ОС Linux используют "хитрый" механизм VDSO (Virtual Dynamic Shared Object). VDSO является небольшой (и неудаляемой) областью памяти, которую ядро операционной системы помещает в адресное пространство пользователя. В этой памяти располагаются набор процедур (небольшая системная библиотека), эти процедуры обеспечивают (как и команды **sysenter**/**syscall**) быстрые (без использования механизма прерывания) системные вызовы. Особо следует отметить, что эти вызовы делаются прямо из программы пользователя, без переключения в привилегированный режим. Таким образом, небольшой (наиболее часто используемый) набор системных вызовов может работать быстрее, чем по командам **sysenter**/**syscall**.

¹ Основные библиотеки Windows:

ntdll.dll – обслуживание системных (Win API) вызовов;
kernel32.dll – функции управления (памятью, файлами, приложениями и т.д.);
user32.dll – функции интерфейса (окна, сообщения, меню и т.д.);
gdi32.dll – функции управления графическим интерфейсом.

² Обычно для экономии команд возврат из **sysenter** (т.е. на метку Cont) заменяют на вход во вспомогательную (библиотечную) процедуру SYSENTER_RETURN, которая выполняет команды:

```
pop  ebp
pop  edx
pop  ecx
ret
```


Необходимо также заметить, что в развитых языках высокого уровня у программистов появилась возможность писать собственные аналоги процедур-обработчиков прерываний. Операционная система предоставляет прикладным программам так называемый механизм обработки структурных исключений SEH (Structured Exception Handling). На базе этого механизма в языках высокого уровня может быть реализован свой механизм обработки исключительных ситуаций, и программа (точнее поток) может "попросить" операционную систему для некоторых *программных* прерываний не начинать выполнение стандартной системной процедуры-обработчика этого прерывания, а передать управление указанной функции в программе пользователя. Обычно говорят, что программа *регистрирует* свою функцию обработки исключений. Эта функция будет выполнять собственные действия по обработке этого события, она может как возобновить (после исправления ошибки), так и аварийно завершить выполнение потока. Таким образом, при возникновении сигнала прерывания системная процедура-обработчик этого сигнала сначала пытается вызвать соответствующую процедуру, написанную программистом, и только если такой процедуры нет, или она не смогла исправить ошибку, то производит стандартную обработку сигнала прерывания.

Аппарат прерываний позволяет реализовать особый стиль программирования, с так называемыми процедурами обратного вызова (call back procedure), в Windows они называются *сервисами*, а в операционной системе Unix они имеют экзотическое название процедуры-демоны.¹ Такая процедура при своей *инициализации* (размещении в памяти) оставляет в определённом месте адрес своего начала. Далее вызов этой процедуры производится из процедур-обработчиков прерываний при возникновении соответствующих условий. Так определяются, например, описанные выше функции обработки структурных исключений, головные функции порождаемых программой потоков (нитей) и др.

В качестве примера рассмотрим расчет платы за междугородний телефонный разговор, при котором за каждую новую минуту разговора к общей сумме прибавляется некоторая величина – тариф за минуту разговора с данным городом. При наступлении льготного периода времени (обычно ночью и в выходные дни) срабатывает будильник (специальная системная программа-обработчик прерываний от встроенного в ЭВМ таймера), который вызывает сервис пересчета тарифа.

Процедуры-демоны широко используются в современных вычислительных системах, например, при приходе электронного письма вызывается системная процедура – почтовый демон и т.д. Все драйверы (программы, управляющие внешними устройствами) тоже являются сервисами. Некоторые сервисы отслеживают работу определённых подсистем (например, обмен с сетью или загрузку процессоров), такие сервисы называются *мониторами*. Всеми сервисами, естественно, управляет менеджер сервисов SCM (Service Control Manager), куда же без начальства 😊.

Заметим, что в некоторые языки высокого уровня включены аналогичные возможности, например, в языке C можно писать так называемые функции-реакции на сигналы со стороны других вычислительных процессов. Заметим также, что очень похожий стиль "событийного" программирования используется и в операционной системе Windows.

Итак, описанный выше механизм обработки прерываний предполагает обработку события как аппаратурой ЭВМ (процессором), так и программой (процедурой-обработчиком прерывания). В качестве аналогии в живой природе выступают, как уже говорилось, безусловный и условный рефлекс живого организма. Существуют, однако, такие события, которые либо просты, либо специфичны, что для реакции на них достаточно, образно говоря, одного безусловного рефлекса.

✚ В современных ЭВМ тоже существуют события, которые обрабатываются по особому, без привлечения аппарата прерываний (точнее, без участия процедуры-обработчика прерывания). Такие сигналы прерываний приходят на специальный вход процессора SM#, они переключают ЭВМ в так называемый режим *системного управления* SMM (System Management Mode). В качестве примера можно привести сигнал аппаратного сброса (hardware reset). Это событие вызывается включением электрического питания или особым сигналом RESET#.iv [см. сноску в конце главы] По этому сигналу процессор производит настройку (или перенастройку) своих аппаратных характеристик (тактовую частоту, режим работы кэш памяти, напряжения питания и т.д.). Иногда, правда, процессор определяет, что сам не может полностью обработать такое событие и, тогда он вызывает

¹ Такое экзотическое название появилось, наверное, потому, что они вызываются не командой **call**, как все "нормальные" процедуры, а начинают выполняться (появляются ниоткуда) как бы "сами по себе", по некоторым событиям, обычно не имеющим отношения к считающейся в данный момент программе.

программу начальной загрузки BIOS или другую специальную программу, которые не являются обработчиками прерывания.

В заключение этого по необходимости краткого рассмотрения прерываний следует заметить, что появление в компьютерах системы прерываний было, несомненно, одним из важнейших событий в развитии архитектуры вычислительных машин. Вспомним, что ничего подобного в рассмотренных ранее учебных машинах не было. Там при возникновении аварийных ситуаций в центральной части машины (деление на ноль, команда с несуществующим кодом операции и т.д.) просто происходил останов машины, а в устройствах ввода/вывода не происходило ничего такого, что могло бы "заинтересовать" центральный процессор. Точнее, если в устройствах ввода/вывода первых ЭВМ происходила аварийная ситуация, например, сбой в работе, то устройство управления просто останавливало машину, и для возобновления работы требовалось вмешательство человека-оператора.

Как Вы вскоре узнаете, появление системы прерываний было одним из необходимых условий, обеспечивающим работу ЭВМ в так называемом мультипрограммном режиме. Недаром появившиеся компьютеры с системой прерываний (и некоторыми другими важными аппаратными особенностями, о которых будет говориться далее) стали относить к следующему, третьему поколению ЭВМ. Подробнее об этом можно прочитать в книгах [1,3].

Вопросы и упражнения

1. Какие устройства относятся к центральной части компьютера, а какие – к периферийной ?
2. Почему при возникновении события чаще всего необходимо прервать выполнение текущей программы ?
3. Какие сигналы прерывания называются внутренними, а какие – внешними ?
4. Чем отличается обычная процедура-обработчик прерывания от привилегированной ?
5. Почему реакцию на некоторые сигналы прерывания надо уметь временно запрещать ?
6. Что такое аппаратная реакция на сигнал прерывания ?
7. Почему обработчику прерывания может понадобиться свой собственный стек ?
8. Что такое вектор дескрипторов прерываний и для чего он нужен ?
9. Что такое переключение задач и для чего оно нужно ?
10. Кто производит программную реакцию на сигнал прерывания ?
11. Для чего программная реакция начинается в режиме с закрытыми прерываниями от внешних устройств ?
12. Что такое минимальная программная реакция на прерывание ?
13. Что такое контекст задачи и как он используется ?
14. В каких случаях прерывание вызывает переключение контекстов ?
15. Может ли выполняемая программа пользователя заметить, что её выполнение прерывалось сигналами прерываний ?
16. Что такое критическая секция программы ?
17. Каково назначение системных вызовов ?
18. Что такое процедура-демон и чем она отличается от обычной процедуры ?

ⁱ Сигнал прерывания не направляется тем ядрам, которые "спят" для энергосбережения. Обычно различают четыре уровней энергосбережения ядра S1-S4:

- S1 – все кэши ядра сброшены, команды не выполняются, но напряжение на процессор подаётся, многие "не важные" устройства выключены;
- S2 – более глубокий "сон", процессор полностью отключён;
- S3 – спящий (Standby) питание подаётся только на оперативную память;
- S4 – режим гибернации (Hibernation), оперативная память сохранена в энергонезависимой памяти.

Для выхода из состояний S3 и S4 можно задать условие, когда ядро просыпается, обычно это сигнал прерывания от кнопки питания, клавиатуры и/или мышки (а иногда и от сетевой карты). Когда все ядра находятся в режимах энергосбережения, то некоторые *мало приоритетные* прерывания могут вообще ждать (в очереди контроллера внешних прерываний), когда "проснётся" какое-либо ядро.

ii Для продвинутых читателей. Зная правила использования стека при работе с процедурами и при обработке прерываний, программист может использовать и свои собственные вспомогательные (локальные) стеки. Например, пусть необходимо часто вызывать процедуру с заголовком

```
procedure P(var X:mas; Len:longword; var Rez:longint);
```

Для передачи параметров, так описанных в секции данных

```
.data  
A dd 1000 dup (?); Массив  
N dd 1000;          Длина массива  
R dd ?;             Результат
```

по соглашению **stdcall**, как известно, необходимо выполнить команды

```
push offset R  
push N  
push offset A  
call P
```

Эти команды заставляют ЭВМ копировать данные в секцию стека (а на самом деле, как Вы скоро узнаете, в кэш данных). Это тяжёлая операция, которой можно избежать, если заранее расположить фактические параметры в секции данных, а затем использовать участок этой секции как новый (временный) стек:

```
.data  
; Временный стек, сначала стековый кадр для  
; неприоритетного прерывания и локальных данных  
; нашей процедуры P, скажем всего 128 байт  
TempS db 128 dup (?)  
; затем параметры, уже как бы переданные в стек  
Params dd offset A;   Адрес массива  
        dd 1000;      Длина массива  
        dd offset Rez; Адрес результата
```

А вот теперь обращение к процедуре P можно производить в виде

```
mov  ebp,esp;   Запомним старый стек  
lea  esp,Params; Переключение на новый стек  
call  P  
mov  esp,ebp;   Восстановление старого стека
```

Внутри процедуры к локальным переменным можно обращаться, используя, вместо регистра ЕВР, регистр ESP, как базовый. Так делают некоторые компиляторы, это позволяет экономить регистр ЕВР, который можно, таким образом, использовать для других целей:

```
; При входе [esp+4] = Params  
push  ebx;      Теперь [esp+8] = Params  
mov   ebx,[esp+12]; Адрес массива  
push  ecx;      Теперь [esp+12] = Params  
mov   ecx,[esp+20]; Длина массива  
; и т.д.
```

При таком вызове процедуры нет необходимости многократно копировать фактические параметры из секции данных в секцию стека, однако здесь есть и несколько существенных особенностей:

1. Процедуру лучше вызывать не по соглашению о связях **stdcall**, а по соглашению **cdecl**, т.е. перед возвратом не очищать стек от параметров.
2. Процедура не должна располагать во временном стеке *команды*, так как, в отличие от настоящего стека, секция данных обычно закрыта на выполнение.
3. Параметры, переданные *по значению* и изменённые в процедуре сохраняют эти изменения после возврата (⚠️), т.е. передаются одновременно как бы и по значению, и по ссылке 🐼.

iii Для продвинутых читателей. При переключении задач в TSS *автоматически* (т.е. процессором) не спасаются регистры вещественной арифметики (регистры арифметического сопроцессора) и векторные регистры, хотя место для них и можно зарезервировать в дополнительной части TSS. Этих регистров очень много и это бы сильно увеличило время переключения между задачами. Следует, однако, заметить, что практически для всех обработчиков прерываний эти регистры не нужны. Исходя из этого, вместо спасения вещественных регистров, процессор устанавливает в управляющем регистре CR0 особый флаг TS "задача переключена" (Task Switched), это запрещает для текущей программы использование команд, работающих с вещественными регистрами.

В том редком случае, когда обработчику прерывания всё же нужны вещественные регистры и он начинает работать с ними при TS=1, то происходит прерывание и обработчик уже этого прерывания спасает (командой

fxsave) регистры сопроцессора в дополнительной части TSS, затем командой **clts** (CLear Task Switched) сбрасывает флаг TS. При возврате из обработчика прерываний, который использовал вещественные регистры, он, естественно, должен их восстановить (командой **fxstor**). Такой способ называется "ленивым" переключением контекста вещественного сопроцессора (Lazy FPU context switching).

Как видно, аппаратная часть работы по переключению задач очень сложна, но всё равно не производит *полного* переключения контекста. Исходя из этого, в 64-битном режиме решено переключение контекста производить полностью программно. Другими словами, процессор аппаратно спасает в (системном) стеке только регистры RIP, RFLAGS, RSP, SS и CS, всё остальное (при необходимости) спасает и восстанавливает процедура обработки прерывания. Для этого предназначены команды **xsave** и **xrstor**, которые спасают весь контекст, включая вещественные и векторные регистры.

Таким образом, мы практически вернулись к схеме переключения контекста, который был старом в реальном режиме. Действительно, в младших (16-битных) моделях семейства Intel аппаратная реакция на сигнал прерывания была значительно проще. В стек прерванной программы записывались регистры FLAFS, CS и IP, в регистре FLAFS сбрасывались флаги IF и TF, затем включался привилегированный режим и производился переход на процедуру-обработчика этого прерывания. Вся работа по сохранению остального контекста прерванной программы ложилась на процедуру-обработчика прерывания. Ясно, что это *программное* переключение контекста работало много медленнее, чем *аппаратное* переключение самим процессором. Однако в 64-битном режиме (полный) контекст потока стал таким большим (достаточно вспомнить о 512-битных векторных регистрах!), что пришлось вернуться в программному сохранению и восстановлению контекста при переключении задач. Как следствие, в 64-битном режиме осталось только два вида шлюзов в таблице дескрипторов прерываний: прерывания и ловушки.

^{iv} Этот сигнал процессор может послать сам себе по специальным командам, а не только получить по кнопке RESET, которая теперь, кстати говоря, теперь часто вообще отсутствует на персональных компьютерах. Аналогичный сигнал процессор может получить и *извне*, через порты ввода, например с сетевой карты. Как видно, реакцию на такие сигналы процессор выполняет даже тогда, когда в его оперативной памяти вообще нет программ, и даже операционной системы (ОС). В режиме SMM процессор выполняет программу, записанную в специальной статической энергонезависимой памяти SMRAM (System Management RAM), она обычно невидима не только для ОС, но и для остальной аппаратуры ЭВМ. Перед переходом в режим SMM в этой памяти сохраняется состояние процессора (без сопроцессора), так что возможен возврат к прерванному состоянию ОС по особой команде RSM, недействительной в других режимах.

В режиме системного управления возможно и переключение на особый (вспомогательный) процессор, который имеет интегрированную в свою микросхему энергонезависимую, не перезаписываемую оперативную память и специализированную ОС (MINIX). Этот процессор тоже невидим для ОС и остальной аппаратуры компьютера, он будет работать даже при тяжёлых аппаратных сбоях, например, при отказе всех процессорных ядер от теплового перегрева, выходе из строя важных шин на материнской плате и т.п. Отметим, что этот процессор может работать даже при выключенном компьютере (если есть напряжение питания, т.е. в ноутбуке, планшете и смартфоне почти всегда).

В процессорах Intel эта *закрытая* технология появилась с 2005 года и называется AMT (Intel Active Management Technology), а все действия по удалённому управлению (даже не работающим!) компьютером осуществляет так называемая Intel Management Engine (Intel ME). Официально эта технология предназначена для тестирования компьютера при его запуске, управления энергопотреблением и удалённой установки нового (или модификации старого) программного обеспечения системными администраторами крупных фирм. Intel ME встроена в микросхему PCH (Platform Controller Hub), на современных материнских платах она исполняет функции южного моста, через неё проходит всё общение с внешними устройствами.

Получается, что в нашем компьютере существует и какой-то "теневого компьютер", который может включаться без нашего ведома (например, по сигналу из сети), причем даже тогда, когда "настоящий" компьютер не работает. Этому теневому компьютеру доступны все аппаратные ресурсы, в частности, диски, а его работа невидима и не контролируется не только штатной ОС, но и остальной аппаратурой материнской платы (например, аппаратными устройствами для аутентификации пользователя).

Принято считать, что ОС и некоторые "крутые" программы (например, отладчики SoftIce и WinDbg) работают в нулевом кольце защиты (Ring 0). *Гипервизор* (Hypervisor) – это программа, эмулирующая работу ЭВМ сразу под управлением нескольких ОС, работает в Ring –1. Это кольцо защиты реализуется технологией под названием VT-X (Virtualisations Technology X), она появилось в процессорах Intel в 2006 году. Под управлением гипервизора (Hypervisor mode), который в фирме Intel называется Монитором виртуальных машин VMM (Virtual Machine Monitor), сама ОС уже не может определить, работает ли она на "настоящей" машине или нет, так как гипервизор полностью контролирует доступ к портам ввода/вывода, служебным регистрам и физической памяти. Разумеется, в таком виртуальном режиме скорость работы ОС падает на 30 и более процентов.

Так вот, что касается "теневого компьютера", то он работает в кольце привилегий Ring –2.

