Глава 7. Система прерываний

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

Конфуций, V век до н.э.

Это сложная тема и Вам надо приложить усилия для её изучения.

Далее продолжим изучение *переходов*, т.е. нарушений естественного порядка выполнения команд программы. По принципу фон Неймана такие переходы могут выполняться только специальными *командами переходов*. Напомним, что в изучаемой нами ЭВМ такие «командные» переходы делятся на безусловные (команды **`jmp`**, call и ret) и условные (команды **`jz`**, **`jg`** и т.д., а также команда **`loop`**).

Современные ЭВМ нарушают этот принцип фон Неймана, в них некоторые переходы производятся не при выполнении *команд* программы, а могут делаться процессором *автоматически* при возникновении определённых условий. Если компьютер обладает такими возможностями, то говорят, что в нём реализована система прерываний (или *аппарат* прерываний). Заметим, что самые первые компьютеры (их относят к ЭВМ первого и, частично, второго поколений) такими возможностями не обладали, однако сейчас все ЭВМ имеют систему прерываний, и Вам предстоит изучить, что это такое.

Необходимо подчеркнуть, что из-за сложности этой темы мы будем рассматривать её только на концептуальном, а не на внутреннем уровне ви́дения архитектуры ЭВМ.

Сначала о терминологии. При запуске на счёт программа (её часто называют *приложение* – application) образует задачу (task в ОС Windows) или процесс (process в ОС семейства Unix), это одно и то же, мы будем использовать эти термины как синонимы.[1] Разные программы порождают разные задачи, однако и одна программа может породить несколько независимых задач. Например, приложение (браузер) Chrome, может порождать несколько задач (отображаемые в окне браузера как вкладки), по одной (а иногда и больше) для каждого открытого сайта.

Сама по себе задачи, как и породившая её программа, пассивна, для своего счёта задача порождает один или несколько (вычислительных) потоков (в Windows) или нитей (в ОС семейства Unix),[2] будем использовать термин поток (понимаемый как поток выполняемых команд). Итак, каждая программа при запуске на счёт порождает хотя бы одну задачу, а каждая задача порождает хотя бы один поток. Поток является минимальной независимой единицей работы операционной системы (для его управляющей программы, обычно называемой диспетчером 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).

Полезно сравнить понятия *программа* (приложение), *задача* и *поток* с аналогичными сущностями машины Тьюринга, которая должна быть Вам хорошо известна по курсу «Алгоритмы и алгоритмические языки». Итак, в машине Тьюринга программа является таблицей (записью алгоритма), находящейся в некотором хранилище информации (на диске 🙂). Когда возникает необходимость выполнить алгоритм, *копия* этой таблицы помещается в некую среду выполнения (оперативную память машины Тьюринга 🙂), порождая таким образом задачу.

Сама по себе задача для машины Тьюринга (это таблица) пассивна (а для компьютера это просто программа, расположенная в сегменте команд). Для выполнения программы надо породить (вычислительный) поток, при этом задаче машины Тьюринга «выдаётся» лента со входным словом и головка, установленная на первую букву этого слова. Кроме того, потоку «выдаётся» переключатель состояний, установленный на первое состояние машины Тьюринга q_1. Соответственно, при порожде-

[1] Например, диспетчер задач Windows в своей таблице называет задачи процессами, такое название осталось с тех времён, когда у каждой задачи был только один процесс. В бóльшей части документации ОС Windows принят термин «задача».

[2] Поток и нить (*жарг.* тред) – это синонимы, два разных перевода английского слова thread. В некоторых книгах их ещё называют *легковесными процессами*.

нии потока на ЭВМ, программе (распооженной в сегменте команд) «выдаются» сегменты данных и стека, а также «экземпляр» всех регистров, там и регистр адреса `EIP` (это как головка машины Тьюринга 🙂).

Задача может порождать несколько потоков. Для машины Тьюринга это несколько лент, по каждой из которых «бегает» своя головка, и к каждой ленте «прилагается» свой переключатель состояний с текущим состоянием q_i, при этом таблица у всех таких потоков одна. Для компьютера это несколько вычислительных потоков, у каждого свои сегменты данных и стека, а также свой набор всех регистров, а вот сегмент команд у них один (общий).

Потоки выполняются *независимо* друг от друга, хотя операционная система и позволяет потокам на компьютере (как правило, в рамках одной задачи) обмениваться между собой данными (это тема курса следующего семестра по операционным системам). «Потоки» в машине Тьюринга, естественно, не могут передавать символы с одной ленты на другую.[1]

Логически потоки выполняются параллельно во времени, хотя возможно, и с разной скоростью. Например, головка на одной ленте сдвигается на один символ за один такт работы «компьютера Тьюринга», а на другой ленте за два такта. Для компьютера это означает, что процессорные ядра, назначенные выполнять потоки, работают с разными тактовыми частотами.

В современных операционных системах поток иногда может порождать несколько волокон (fiber), их ещё называют облегчёнными процессами. У каждого волокна одного потока сегменты команд и данных общие, но свой стек и «экземпляр» регистров, однако два волокна одного потока не могут работать параллельно (одновременно на двух ядрах). Для диспетчера задач операционной системы волокна невидимы, они сами, в рамках одного потока «договариваются» о переключении выполнения с одного волокна на другое. Для машины Тьюринга это означает, что у ленты потока не одна головка, а несколько 🙄, правда, эти головки не могут двигаться одновременно, а только по очереди, «договариваясь» между собой о порядке своих движений. Запрет параллельной работы волокон и одновременного движении головок существенен. Вам необходимо понять, что если его снять, то у алгоритма пропадает свойство *детерминированности*, и, он перестаёт быть алгоритмом 😱. Волокна в программировании используются редко, их единственное преимущество перед потоками в том, что переключение компьютерного ядра с одного волокна на другое происходит быстрее, чем с одного потока на другой.

Теперь введём понятие события (возникшей ситуации), которая может нарушить естественный порядок выполнения команд программы. Событие может касаться как собственно выполняемого на ЭВМ потока команд (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 (и далеко не всегда «нехороших») команд программы, т.е. *синхронно* с командами программы.

Внешние (асинхронные) события могут возникать в периферийных устройствах. Например, нажата или отпущена 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 или увеличить тактовую частоту) и др.

Как видим, внешние события достаточно примитивны, например, для клавиатуры это нажатие и отпускание кнопки, а всю «интеллектуальную» работу по обработке такого внешнего события выполняет программа-драйвер. Например, если нажата некоторая кнопка клавиатуры, драйвер засекает время этого события, после чего «время от времени смотрит на часы». В случае, если кнопку не отпускают, скажем, больше секунды, драйвер начинает, например, каждую пятую долю секунда ставить в очередь `input` по одному символу, соответствующему нажатой кнопке (автоповтор).

Чаще всего бывает так, что при возникновении некоторого события продолжать выполнение текущего программного потока либо бессмысленно (несуществующий код операции), либо нежелательно, так как нужно срочно предпринять какие-то действия, для выполнения которых этот поток

[1] Можно построить и такую, более сложную версию машины Тьюринга, с «общающимися» лентами, впрочем, её тоже может эмулировать универсальная машина Тьюринга.

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 прерываний (Interrupt Service Routine, Interrupt Handler). Как мы уже говорили, обработчики событий на внешних (периферийных) устройствах называются драйверами (driver – водитель, кучер 🙂).

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 много тысяч раз в секунду.

В архитектуре современных компьютеров предусмотрено, что каждое устройство, в котором произошло событие (процессор, оперативная память, устройство ввода/вывода) генерирует при этом особый сигнал прерывания (Interrupt Request) – электрический импульс, который приходит на специальную электронную схему процессора, которая носит название контроллер прерываний. 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, контроллер записывает его в свою очередь [1] пришедших сигналов для последующей обработки этого сигнала процессором, который 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 «спрашивает» у контроллера, пришел ли сигнал прерывания, только после выполнения очередной команды, таким образом, этот сигнал ждёт *завершения* текущей команды.

Разумеется, некоторые команды не могут быть выполнены до конца. Это, например, целочисленное деление на ноль или обращение к отсутствующей странице виртуальной памяти (об этом будет подробно говориться в курсе по операционным системам). Обычно в этом случае, как будет говориться далее, процессор возвращает счётчик адреса, снова установленный на начало такой «нехорошей» команды, чтобы её можно было, при необходимости, повторить.

Отметим также, что перед (основной) командой могут располагаться вспомогательные команды-префиксы, например, мы хорошо знаем префикс `66h`. Префиксы не считаются «настоящими» командами, после их выполнения сигнал прерывания не анализируется, Исключения составляют префиксы цикла **`rep`**, **`repe`** и **`repne`** (о них будет рассказано в другой главе), после выполнения этих команд-перфиксов прерывания допускаются.

Здесь также надо упомянуть о «нетипичной» команде в архитектуре рассматриваемого компьютера, это команда с кодом операций **`HLT`** (HaLT). Эта команда останавливает выборку процессором других команд, 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, в этом режиме процессор почти не потребляет энергии.

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

---

[1] Как будет рассказано далее, это не совсем очередь, а скорее список, так как сигналы выбираются оттуда на обработку не в порядке поступления, а по их приоритету. Всего у контроллера (внешних) прерывание 8 входов, `IRQ0-IRQ7`, приоритет прерывания падает с увеличением номера входа.

конвейерные ЭВМ, они могут одновременно выполнять несколько десятков команд. Для конвейерных ЭВМ необходимо уточнить, когда начинается аппаратная реакция на сигнал прерывания. В каждый момент времени (такт работы) одна или несколько команд могут быть полностью выполнены (как говорят, «уйти в отставку», см. разд. 14.2.1). Выполнение остальных (частично выполненных) 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).

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 счёта задач. Более подробно о конвейерных ЭВМ будет говориться в другой главе этой книги.

Итак, сигналы прерываний приходят на специальную электронную схему ЦП, обычно она называется (программируемым) контроллером прерываний APIC (Advanced Programmable Interrupt Controller).[1] 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 каждое ядро делит на ноль независимо от других ядер 🙂), и один на все ядра контроллер *внешних* прерываний (APIC IO). При этом каждое процессорное ядро после выполнения очередной команды «смотрит» на свой контроллер внутренних прерываний, а контроллер внешних прерываний сам посылает сигнал прерывания одному из внутренних контроллеров. По умолчанию сигнал прерывания с такого общего внешнего контроллера приходит к внутреннему контроллеру ядра с номером ноль. Можно, однако, задать всем ядрам аппаратные приоритеты, и тогда прерывание приходит к ядру, которое сейчас работает с наименьшим приоритетом.[2] 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 ядру.

Сигнал прерывания не направляется тем ядрам, которые «спят» для энергосбережения. Обычно различают четыре уровней энергосбережения при работе 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», ядро полностью отключёно;
- S3 – спящий (Standby), все ядра отключены, питание подаётся только на оперативную память и контроллеры прерываний;
- S4 – гибернация (Hibernation), оперативная память сохранена в энергонезависимой памяти и питание на неё больше не подаётся, работают только контроллеры прерываний.

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.

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. Когда в очереди стоят несколько сигналов прерывания, процессор выбирает на обработку наиболее приоритетный из них. Способы установки таких приоритетов здесь обсуждаться не будут, это тема курса по операционным системам.для некоторых из этих при-

---

[1] Первая отечественная ЭВМ М40 с (очень примитивной) системой прерываний появилась в 1958 году. Первый Programmable Interrupt Controller (PIC) Intel 8229 для *персональных* ЭВМ появился в 1978 году.

[2] Различают аппаратный приоритет ядра IRQL (Interrupt ReQuest Level) и приоритет выполняемого на этом ядре в данный момент вычислительного потока, приоритет потока назначается операционной системой.

шедших сигналов они *игнорируются*, и ядро продолжает выполнение команд текущего потока. Говорят, что прерывания с такими номерами в данный момент времени запрещены или *замаски́рованы*.

Термин «замаски́ровать» появился потому, что разрешение прерывания обычно задаётся установкой в единицу определенного бита в некотором служебном регистре. На первых внешних носителях данных двоичная единица обозначалась отверстием (прорезью) в нужном месте бумажной перфоленты или 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. *замаски́рованным* (на него установлена маска, у которой нет прорезей для глаз). Надо отметить, что у фирмы Intel нет единого подхода к трактовке «единичной» маски. Так в регистре состояния и управления векторными операциями `mxcsr` единичная маска в бите, наоборот, *запрещает* наступление события, связанного с этой маской.

Для компьютера нашей архитектуры можно замаски́ровать 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 замаски́ровать каждое прерывание от внешних устройств по отдельности, для этого надо установить в ноль соответствующий этому прерыванию бит в *регистре маски* прерываний `IMR` контроллера прерываний. В принципе, и прерывание с №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.

Маски́ровать 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, только внешние прерывания с определёнными номерами можно закрывать (маски́ровать) и открывать (разрешать, снимать с них маску).

В том случае, если прерывание игнорируется (замаски́ровано), сигнал прерывания, тем не менее, продолжает оставаться в очереди контроллера внешних прерываний до тех пор, пока маски́рование этого сигнала не будет снято, или же на этот контроллер ни придёт следующий сигнал прерывания. В последнем случае первый сигнал может безвозвратно потеряется, что опасно, так как компьютер не прореагировал должным образом на некоторое событие, и оно прошло для него *незамеченным*. Чтобы снизить такую опасность, контроллер прерываний, как уже упоминалось, обычно запоминает приходящие сигналы в небольшой аппаратно реализованной кольцевой *очереди* для их дальнейшей обработки. Следовательно, для надёжной работы маски́ровать 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 маски́ровать некоторые внешние прерывания. Дело в том, что, если пришедший сигнал внешнего прерывания не замаски́рован, то, как вскоре станет ясно, процессор прекращает выполнение *текущего* потока и переключается на выполнение *другого* потока, в котором находится процедура-обработчик данного прерывания ISR (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 или важным внешним устройством (линия связи с космическим аппаратом, ядерный или химический реактор и т.д.).

С другой стороны, необходимо понять, что должны существовать и специальные *немаски́руемые* сигналы прерывания от внешних устройств (Nonmaskable External Interrupts). В качестве примеров 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 счёт 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 энергонезависимой (Nonvolatile) памяти , переключиться на резервный блок питания и т.д. Как уже говорилось, нельзя маски́ровать 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 неко-

торых специальных команд, основным назначением которых как раз и является выдача сигнала прерывания (вскоре эти команды будут изучены). Для данной архитектуры все сигналы о событиях в центральной части компьютера замаски́ровать нельзя, и, как уже упоминалось, существует только одно немаски́руемое прерывание от *внешних устройств* №2.

При описании архитектуры процессоров Intel x86 внутренние прерывания называются исключениями (exceptions), имеется в виду исключительная ситуация, возникшая при выполнении команды программы, 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 текущего потока.

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

- Нарушение (fault – ошибка, отказ) можно определить до фактического завершения выполнения команды, поэтому счётчик адреса `EIP` будет указывать не на следующую команду (как обычно), а на *текущую*, которая вызвала это нарушение, при этом флаг `RF` (Restart/Resume Flag) повторения (рестарта) команды в регистре `EFLAGS` установлен в 1. Таким образом, эту команду можно при необходимости *повторить* (если, конечно, ошибку удалось исправить). В качестве примера можно привести 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 (строковые команды описаны в разд. 8.1).
- Ловушкой (trap) называется исключение, при котором текущая команда полностью выполнена и счётчик адреса указывает, как обычно, на *следующую* команду и можно при необходимости *продолжить* выполнение прерванного программного потока. Так, например, работает отладчик при останове в контрольных точках или выполнении потока 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 прерванного потока со следующей команды.
- Авария (abort) 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 прерванного потока невозможно, он должен принудительно завершиться операционной системой.

Продолжим теперь рассмотрение аппаратной реакции на *незамаски́рованное* прерывание. Такая реакция сильно зависит от *режима* работы компьютера. Компьютеры рассматриваемой архитектуры могут работать в нескольких режимах. Это так называемый *реальный* режим (Real Mode), режим *виртуальном 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.

Особым является случай, когда во время прихода незамаски́рованного сигнала прерывания компьютер занят обработкой другого сигнала прерывания (его номер храниться на регистре обслуживания прерывания `ISR`). В этом случае сравниваются приоритеты обрабатываемого и нового сигналов прерываний, если приоритет вновь пришедшего сигнала не выше обрабатываемого, то новый сигнал ставится в очередь, иначе поступает на обработку процессором.

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 (см. разд. 6.11.1) и может разделять его ресурсы, в частности, использовать его стек. В качестве примера такой процедуры-обработчика мож-

но 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. Как уже говорилось в разд. 6.11.1, 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.

7.1. Таблица дескрипторов прерываний

Усердие всё превозмогает!
Козьма Прутков

Поражение неминуемо ждет лишь того, кто отчаялся заранее.
Джон Рональд Руэл Толкин
«Властелин колец»

Каждый номер прерывания определяет для процессора так называемый *дескриптор* (описатель) процедуры-обработчика прерывания с данным номером. Все такие дескрипторы (каждый длиной по 8 байт) располагаются в специальной таблице (массиве) дескрипторов прерываний с именем `IDT` (Interrupt Descriptor Table), максимальная длина этой таблицы `8*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 памяти, а не обязательно с нулевого адреса, как было в младших моделях первого и второго поколений процессоров семейства Intel. Для многоядерных ЭВМ у каждого процессорного ядра свой регистр `IDTR` и, соответственно, своя `IDT`.

При описании архитектуры рассматриваемых ЭВМ дескрипторы процедур-обработчиков прерываний принято называть шлюзами (Gate), они «ведут» к процедурам-обработчикам прерывания. Заметим, что, при проходе через шлюз на реке, судно, например, поднимается в шлюзе и выходит в реку выше по течению. Так и при вызове через шлюз может запуститься программа с более высоким приоритетом, а при возврате на прерванную программу (обратный проход по шлюзу) приоритет снова опускается.

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 вызываемой процедуры, если этот уровень выше привилегий текущего потока, т.е. `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 выполняемого потока и системный стек (после малого упрятывания) для обычного и приоритетного прерывания соответственно. Индекс `P` у регистра обозначает регистр прерванного потока, а индекс `I` – регистр процедуры-обработчика прерывания. Отметим, что для

[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.

приоритетного прерывания используется только системный стек, а стек прерванной программы не меняется.

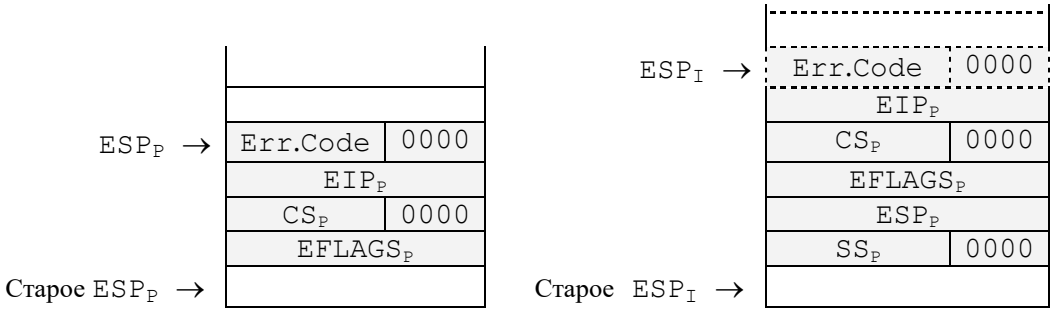

Рис. 7.1. Вид стека программы и системного стека после малого упрятывания.

Как видно, для текущего потока в стек записывается регистр флагов EFLAGS, регистры EIP и CS для обеспечения возможности возврата на следующую команду прерванного потока, а для некоторых исключений также (необязательный) код 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 уже говорилось запомненный в стеке счётчик адреса EIP_P может указывать как на следующую, так и на текущую команду прерванной программы. Как видим, эти действия похожи на действия при выполнении команды дальнего вызова процедуры через шлюз (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.[i] [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 (см. разд. 13.1) и в своём адресном пространстве, в частности, со своим стеком.[1]

При переключении задач процессор для прерванного потока автоматически запоминает информацию, необходимую для возобновления его счёта. Необходимо запомнить все сегментные регистры и регистры общего назначения, 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 потока (или контекстом *задачи*).

У первых операционных систем каждая задача могла порождать только один поток, поэтому исторически часто поток и порождающая его задача воспринимаются как синонимы, именно поэтому контекст потока у нас часто называется контекстом задачи, а механизм автоматического переключения потоков называется у нас переключением задач.

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). Каждый TSS определяется своим дескриптором, все такие дескрипторы хранятся в глобальной дескрипторной таблице GDT. Селектор (номер) TSS в GTD 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

[1] 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 (Task State Segment).

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.

TSS считается *специальным* сегментом памяти, это задаётся битом S=0 (системный объект) в в его дескрипторе. Как следствие, селектор TSS нельзя загрузить в сегментный регистр (только в TR), и никакая программ (даже с нулевым уровнем привилегий), не может читать и писать в этот сегмент, доступ к нему имеет только аппаратура процессора. 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 системы создаёт *псевдоним* этого сегмента (как сегмент данных), чтобы заполнить TSS нужными значениями, затем этот псевдоним уничтожается. Всё это показывает степень защиты контекстов задач.

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 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.[ii] [см. сноску в конце главы] Вот теперь аппаратная реакция на привилегированное прерывание закончилась и начинает выполняться (привилегированная) процедура-обработчик данного прерывания.

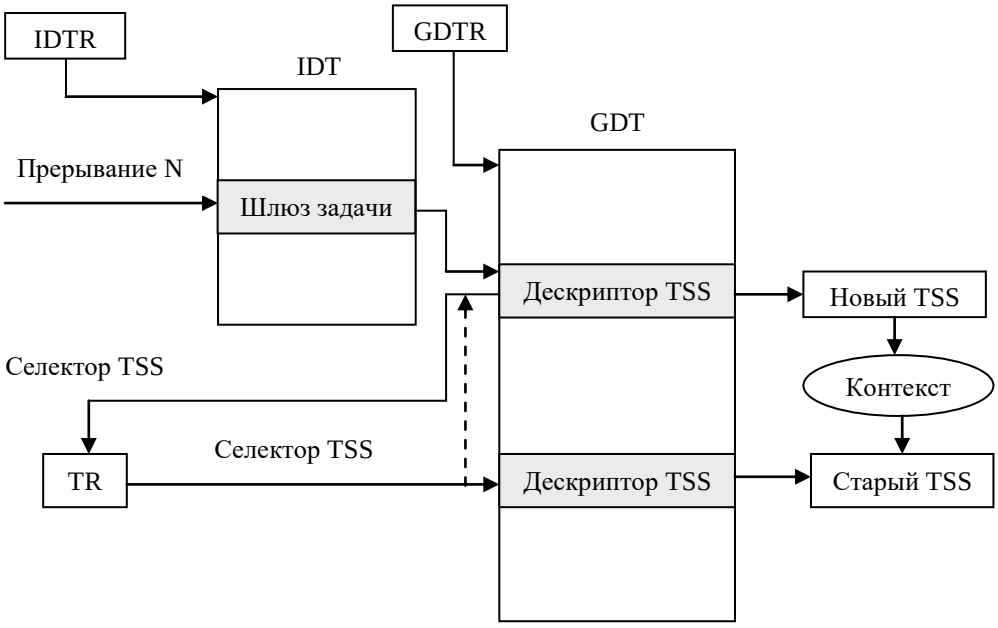

Рис. 7.2. Схема переключения задач для приоритетного сигнала прерывания (IDTR – Регистр таблицы дескрипторов прерываний, GDTR – Регистр главной дескрипторной таблицы, TR – Регистр (текущей) задачи (потока), IDT – Таблица дескрипторов прерываний, GDT – Глобальная таблица дескрипторов).

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 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. Отметим, что 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 (перед выполнения первой же команды) прервана другим внешним сигналом прерывания. Как уже говорилось, значение флага `IF=0` маски́рует все прерывания от внешних устройств, кроме прерывания с №2. Флаг `TF=0` устанавливается потому, что при значении `TF=1` процессор всегда посылает *сам себе* (немаски́руемый) сигнал об исключении EXCEPTION_SINGLE_STEP с номером `N=1` после выполнения *каждой* команды. 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 незамаски́рованное прерывание заканчивается.

Заметим, что после некоторых «хитрых» команд, прерывание 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.

Заметим, что в живой природе некоторым аналогом аппаратной реакции ЭВМ на прерывание 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. Очевидно, 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).

На этом *аппаратная* реакция на незамаски́рованное прерывание заканчивается, и процедура-обработчик прерывания начинает выполнять программную реакцию на прерывание, начинает работать процедура-обработчик прерывания с данным номером.

7.1.1. Схема работы процедуры-обработчика прерывания

Читать, не размышляя, всё равно, что есть и не переваривать.

Эдмунд Берк

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, од-

нако, сохраняется, так как специальное поле PTL в TSS позволяет возвращаться в предыдущий (прерванный) программный поток по команде **iret**.

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.

Напомним, что эта процедура чаще всего начинает 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 **sti** (SeT Interrupt, `IF:=1`).

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 общего назначения, 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 – End Of Interrupt), это удаляет сигнал прерывания из очереди контроллера.

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. Далее производится полная программная реакция на прерывания, т.е. процедура-обработчик выполняет *все* необходимые действия, связанные с происшедшим событием. Обычно для выполнения полной программной реакции порождается новый поток, так называемая отложенная процедура DPC (Deferred Procedure Call). Приоритет этой процедуры (Dispatch Level) выше, чем у обычных прикладных задач (Passive Level) но ниже, чем у процедур, выполняющих минимальную программную реакцию.

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 – повторно входимый). 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.

> 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 этого события.

7.1.2. Критические секции программы

Думать – самая трудная работа; вот, вероятно, почему этим занимаются столь не многие.

Генри Форд

Важным является то, 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, пока один программный поток не вышел из критической секции, другой поток не смог бы туда войти.

На старых ЭВМ проще всего этого можно было достичь, если при выполнении критической секции работа потока не прерывается. Для этого в начале каждой такой секции можно поставить команду 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. Сейчас 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 ⚠). Кроме того, при работе в защищенном режиме Windows пользовательская программа и процедуры-обработчики *неприоритетных* прерываний не могут сами закрывать и открывать внешние прерывания командами **cli** и **sti**. Они вынуждены использовать предназначенные для этого системные вызовы EnterCriticalSection и LeaveCriticalSection, это вызова обеспечивают такую, более сложную защиту критических секций.

Критические секции можно использовать только для синхронизации потоков внутри одной задачи (процесса), для синхронизации независимых задач (процессов) используются другие средства (семафоры, мьютексы и т.д.), эта тема изучается в курсе по операционным системам. Запрет прерывания, по существу является механизмом, не позволяющим другим программным потокам войти в критическую секцию, пока из неё не вышел первый поток. Упомянутые выше другие механизмы доступа к разделяемым ресурсам позволяют потоку добровольно «спрашивать», можно ли входить в критическую секцию и ждать разрешения, если вход в секцию закрыт.

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.

При возврате из привилегированного прерывания различают два случая. Когда переключение на новую задачу произведено с возможностью возврата (с заполнением поля PTL селектором TSS предыдущего 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 возобновления их счёта будет принимать системная программа – *диспетчер задач* (Task Manager).

Диспетчер задач 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 Windows *процессами*). Для каждой задачи диспетчер в отдельной колонке показывает число порождённых ей потоков. При

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

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, для пользователя 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. Более полно 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.

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)	Нарушение или Ловушка
N=2	Немаски́руемое 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)	Нарушение
N=7	Сейчас не используется (математический сопроцессор)	
N=8	Двойная ошибка (#DF – Double Fault), при обработке исключения возникло ещё одно исключение [1]	Авария
N=9	Зарезервировано	
N=10	Плохой TSS (#TS)	Нарушение
N=11	Отсутствие сегмента в оперативной памяти (#NP)	Нарушение
N=12	Выход за границы стека для стековых операций (#SS)	Нарушение
N=13	Ошибка общей зашиты (#GP – General-Protection Exception) – здесь много “нехороших” случаев.	Нарушение
N=14	Нарушение доступа к странице виртуальной памяти (#PF – Page Fault)	Нарушение
N=16	Ошибка при операциях с вещественными числами (#MF)	Нарушение
N=17	Контроль выравнивания (#AC): магическое условие: CR0[AM]=1&EFLAGS[AC]=1&CPL=3	Нарушение
N=18	Ошибка машинного контроля (#MC – Machine Check)	Авария
N=19	Ошибка векторного регистра (#XM)	Нарушение

7.3. Команды прерывания

Поражение неминуемо ждет лишь того,
кто отчаялся заранее.
Джон Рональд Руэл Толкин
«Властелин колец»

Продолжим теперь изучение специальных *команд* прерываний, выполнение каждой такой команды *всегда* вызывает прерывание (исключением является команда **into**, которая вызывает пре-

[1] Это исключение возникает только для пары так называемых "тяжёлых" исключений с номерами 0, 10-13, или для исключения с номером 14 и одного из тяжёлых. В остальных случаях исключения ставяться в очередь для последовательной обработки. А вот когда при попытке вызвать обработчик двойной ошибки возникает любое другое исключение, то процессор реагирует, как на это, как на выполнение команды HLT.

рывание *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).

В современных процессорах есть специальные отладочные регистры, адреса в которых задают контрольные точки и в командах, и в данных (как по записи, так и по чтению), если этих регистров хватает (а у нас их всего 4: `DR0-DR3`, плюс 4 служебных `DR4-DR7`), то команды **int** в программу отладчикам можно не ставить.

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

При надежном программировании проверку флага переполнения необходимо ставить после каждой такой команды. Для такой проверки можно использовать команду **into**, так как это самая короткая (однобайтная) команда условного перехода по значению `OF=1` (обычная команда `jc Error` при дальнем переходе занимает 5 байт). При этом, правда, обработку аварийной ситуации должна производить процедура-обработчик прерывания с номером `N=4`. Отметим, что в 64-битном режиме команда **into** больше не работает. Впрочем, команда условного перехода **jo** теперь работает весьма эффективно, так как задаваемый ей переход практически никогда не выбирается предсказателем перехода в качестве пути выполнения программы (см. разд. 14.2).

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:

```
N equ 1000
.data
Mas dd N dup(?)
Bnd dd Mas,Bnd-4; Границы массива
; можно Bnd dd Mas,Mas+4*N-4
.code
  mov   eax,offset Mas
  bound eax,Bnd; Всё хорошо, Mas[1]
  mov   eax,offset Bnd; За границами, Mas[N+1]
  bound eax,Bnd; Нарушение STATUS_ARRAY_BOUND_EXCEEDED
```

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 можно задать LB (Lower Bound) и UB (Upper Bound) границы контролируемого доступа к памяти. Есть и регистр состояния BNDSTATUS и новые команды для работы с этими регистрами. Поэтому старая команда **bound** больше не работает.

Прерывание N=6 имеет мнемонику #UD (UNdefined) что намекает на особую команду без параметров с кодом операции **ud2** (код 0Fh 0Bh, в Ассемблере MASM-6.14 отсутствует), 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 **s[et]alc** 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.

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

```
int op1=N
```

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, бóльшими 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 передаются на регистрах и никогда в стеке.

Первые BIOS, хранящиеся в памяти типа ROM, 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 фиксированные регистры)

производится быстрее. При этом, однако, многие вещи являются невозможными, например, рекурсивный вызов таких процедур. После 2007 года ЭВМ вместо BIOS снабжаются 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.

Современные операционные системы (в частности, Windows) запрещают программам обычных пользователей (работающих в непривилегированном режиме) применять 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. Системный 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 (этот вызов медленный, около 1000 тактов) рекомендуется использовать новую команду быстрого системного вызова **sysenter**, (в 64-битном режиме она называется **syscall**), этот вызов может быть выполнен 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]

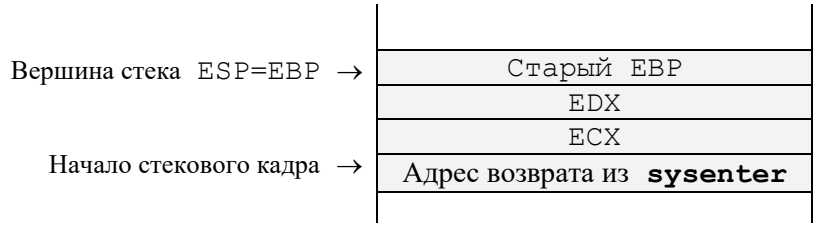

Рис. 7.3. Вид стекового кадра пользователя для команды **sysenter**.

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
```

---

[1] В противовес команде **int** 2Eh ОС Windows, доступной только в привилегированном режиме, соответствующая команда для системного вызова **int** 80h ОС семейства UNIX может использоваться и в режиме обычного пользователя.

[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, возврат результата на RAX.

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
```

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), которая, в свою очередь, содержит внутри себя команду **sysenter**, ведущую в процедуру-обработчик прерывания (обычно в ядре операционной системы). Отметим основные библиотеки ОС Windows:

```
ntdll.dll     – обслуживание системных (Win API) вызовов;
kernal32.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.
```

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 регистров [1]
```

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

```
; 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 Linyx используют «хитрый» механизм VDSO (Virtual Dynamic Shared Jbject). VDSO является небольшой (и не удаляемой) 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**.

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 расчёт 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 пересчёта 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 про-

---

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

[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.

цессоров), 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# [iii] [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 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

Никогда не отчаивайтесь. А если вы уже впали в отчаяние, то продолжайте работать и в отчаянии.

Эдмунд Берк

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 понадобится 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] Для продвинутых читателей. Зная правила использования стека при работе с процедурами и при обработке прерываний, программист на Ассемблере 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 «запас» в стековом кадре для
; неприоритетного прерывания и локальных данных нашей
; процедуры P, скажем всего 128 байт
TempS  db 128 dup (?)
; затем параметры, как бы уже 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;   Переключим на новый стек
  call P
  mov  esp,ebp;      Восстановим старый стек
```

Вообще говоря, внутри процедуры к локальным переменным можно обращаться, используя, вместо регистра EBP, регистр ESP, как базовый. Так делают некоторые компиляторы, это позволяет использовать регистр EBP для других целей, например:

```
; 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 надо 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 🙄.

[ii] 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.

[iii] Для продвинутых читателей. Этот сигнал процессор может послать сам себе по специальным командам, а не только получить по кнопке RESET. 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 (знаменитую 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. Этот невидимый процессор включается, как только подаётся электрическое питание, т.е. 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 CS(M)E (Converged Security (and Management) Engine). 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 CS(M)E, это набор микропрограмм, встроенных в микросхему PCH (Platform Controller Hub), на современных материнских платах она исполняет функции южного моста, через неё проходит всё общение с внешними устройствами. Аналогичная «машина» Platform Security Processor (PSP) создана фирмой AMD и на платформе Андроид (Android TEE).

Получается, что в нашем компьютере существует и какой-то «теневой компьютер», который может включаться без нашего ведома (например, по сигналу из сети), причем даже тогда, когда «настоящий» компьютер не работает и даже не имеет в памяти ни одной программы! Этому теневому компьютеру доступны все аппаратные ресурсы, в частности, диски, а его работа невидима и не контролируется не только штатной ОС и антивирусными программами, но и всей аппаратурой материнской платы (например, аппаратными устройствами для аутентификации пользователя). В сети существуют примеры активизации этой системы обычными пользователями.

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 (минус один). 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 (минус два). Спокойствие, только спокойствие! 🙂.