Обработка прерываний и исключений

Прерывания и исключения – это состояния в ОС, переключающие процессор на код, лежащий вне нормального потока команд. Они обнаруживаются аппаратно и программно. При обнаружении прерывания или исключения процессор прекращает выполнять текущее действие и передает управление в особое место памяти – по адресу кода, обрабатывающего возникшее состояние. В *NT* этот код называется обработчиком ловушки (*trap handler*).

Ядро *NT* различает *прерывания* и *исключения*. *Прерывание* (*interrupt*) – это асинхронное событие, которое может произойти в любой момент времени, независимо от того, чем занят процессор. Прерывания генерируются устройствами ввода/вывода и таймерами процессора и могут быть разрешены (включены) или запрещены (отключены).

Исключение (*exception*) – это синхронное состояние, возникшее в результате выполнения некоторой инструкции. Исключения воспроизводятся посредством повторного выполнения той же программы, с теми же данными и в тех же условиях. Примеры исключений – нарушение защиты памяти, некоторые отладочные команды и ошибки деления на ноль. Ядро *NT* рассматривает как исключения вызовы системных сервисов (хотя технически это системные ловушки).

Обработчик ловушки. Термином ловушка (*trap*) обозначается механизм, используемый процессором (при возникновении в исполняющем потоке прерывания или исключения) для перехвата управления, переключения из пользовательского режима в режим ядра и передачи управления в фиксированную точку ОС. В *Windows NT* процессор передает управление обработчику ловушки ядра *NT*. Этот модуль играет роль коммутационной панели; он принимает исключения и прерывания, генерируемые процессором, и передает управление коду обработки соответствующей ситуации.

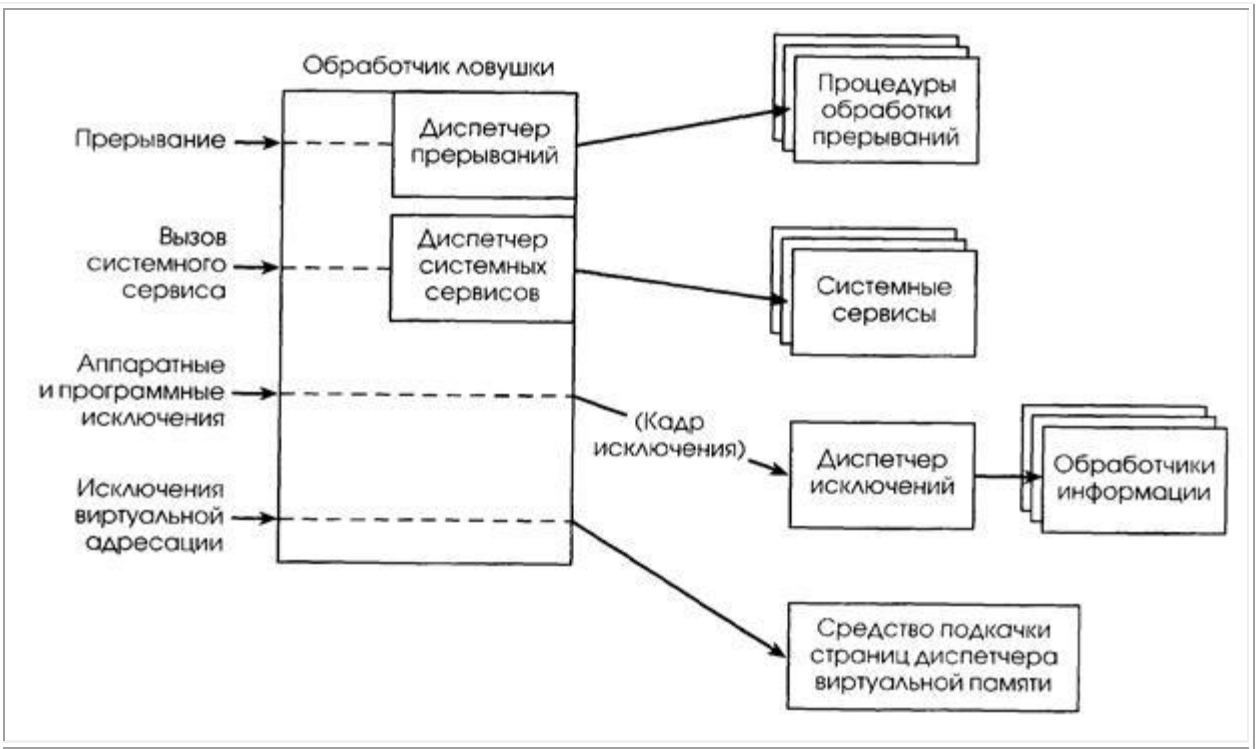

Рис. 2.13. Распределение прерываний и исключений

Прерывание генерируются как устройством ввода/вывода, так и ядро может вызвать программное прерывание. На рис. 2.13 показаны условия, приводящие к активизации обработчика ловушки, и модули, которые он вызывает для обработки этих условий. В момент своего вызова обработчик ловушки запрещает прерывания и сохраняет состояние машины. Он создает кадр ловушки (*trap frame*), в который помещает информацию о состоянии исполнения прерванного потока. Эта информация позволит ядру возобновить исполнение потока после обработки прерывания или исключения. Кадр ловушки является подмножеством полного контекста потока.

Если произошло прерывание от устройства, то управление передается процедуре обработки прерывания (*interrupt service routine, ISR*), предоставляемой драйвером данного устройства. Если прерывание возникло в результате вызова системного сервиса, то обработчик ловушки передает управление коду системного сервиса в исполнительной системе *NT*. Остальные исключения обрабатываются собственным диспетчером исключений ядра.

Распределение прерываний. Прерывания, сгенерированные аппаратно, исходят от устройств ввода/вывода, которые должны уведомлять процессор о необходимости их обслуживания. Устройства, способные генерировать прерывания, позволяют ОС обеспечить максимальную загрузку процессора путем совмещения во времени вычислений и операций ввода/вывода. Процессор запускает операцию ввода/вывода на устройстве и выполняет другие потоки, пока осуществляется пересылка данных. Когда устройство закончило обмен, оно генерирует прерывание для выполнения обслуживания. Координатные устройства, принтеры, клавиатуры, диски и сетевые карты обычно генерируют прерывания. Системное программное обеспечение также может генерировать прерывания.

На прерывания реагирует подмодуль обработчика ловушки ядра, диспетчер прерываний. Он определяет источник прерывания и передает управление либо внешней процедуре обработки прерываний (*ISR*), либо внутренней процедуре ядра. Драйверы устройств предоставляют *ISR* для обслуживания прерываний от устройств, а ядро содержит процедуры обслуживания других типов прерываний.

Типы и приоритеты прерываний. Разные процессоры могут распознавать разное количество и разные типы прерываний. Диспетчер прерываний отображает аппаратные уровни прерываний в стандартный набор уровней прерываний (*interrupt request level, IRQL*), распознаваемых ОС. Уровни *IRQL* ранжируют прерывания по приоритетности. Если приоритет планирования – это атрибут потока, то *IRQL* – атрибут источника прерываний. Кроме того, каждый процессор имеет текущий *IRQL*, который изменяется в процессе работы ОС. Поток, работающий в режиме ядра, может повышать или понижать текущий *IRQL* процессора, на котором выполняется, чтобы замаскировать или демаскировать прерывания нижних уровней.

Ядро определяет набор переносимых *IRQL*, который может модифицироваться, если у данного процессора есть особые возможности, связанные с прерываниями (например, второй таймер). Прерывания обслуживаются в порядке приоритета. Уровни *IRQL* с самого старшего по первый *IRQL* устройства зарезервированы для аппаратных прерываний; прерывания уровней диспетчерский/*DPC* и *APC* – это программные прерывания, генерируемые ядром. Низший *IRQL* на самом деле не является уровнем прерываний – он относится к нормальному выполнению потока, при котором все прерывания разрешены.

Текущий *IRQL* процессора определяет, какие прерывания получает данный процессор. В процессе выполнения поток режима ядра повышает или понижает процессорный *IRQL*. Как показано на рис. 2.14, источники прерываний, уровень которых выше текущего, прерывают работу процессора, в то время как прерывания от источников с *IRQL*, равным или меньшим, чем текущий, блокируются или маскируются (*masked*), пока выполняющийся поток не понизит *IRQL*.

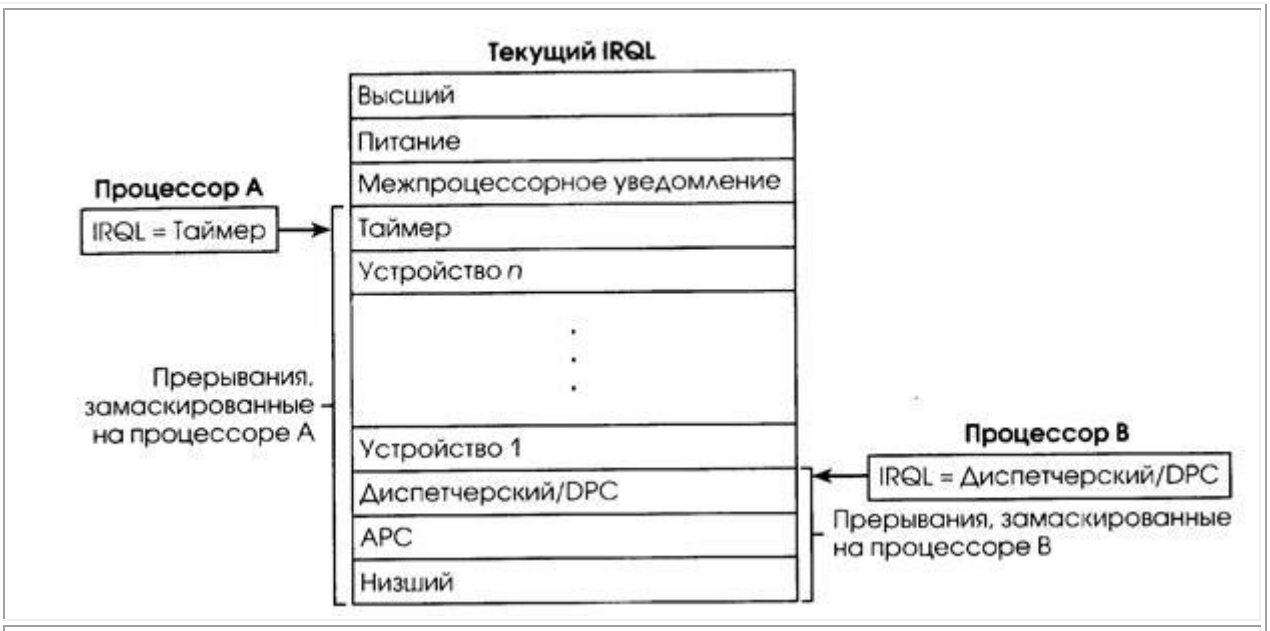

Рис. 2.14. Маскирование прерываний

Поток режима ядра повышает *IRQL* процессора, на котором он исполняется, в зависимости от того, что он в данный момент пытается сделать. Например, когда происходит прерывание, обработчик ловушки (или, возможно, процессор) понижает *IRQL* процессора до уровня, назначенного источнику прерывания. Это блокирует все прерывания данного и низших уровней (только на этом процессоре) и гарантирует, что обслуживающий прерывание процессор не будет перехвачен менее важным прерыванием. Замаскированные прерывания либо обслуживаются другим процессором, либо задерживаются, пока *IRQL* не снизится. Изменение *IRQL* процессора – это мощная операция, которую следует выполнять с большой осторожностью. Потоки пользовательского режима не могут изменять *IRQL* процессора.

Каждый уровень прерываний имеет особое назначение. Ядро генерирует межпроцессорное прерывание (*IPI*), чтобы запросить у другого процессора выполнение действия, например, чтобы направить некоторый поток на исполнение или обновить кэш справочного буфера трансляции (*TLB*). Системный таймер генерирует прерывания через заданные интервалы времени, и ядро обрабатывает их, обновляя текущее значение часов и замеряя время выполнения потока. Если процессор поддерживает два таймера, то ядро добавляет еще один уровень таймерных прерываний для измерения производительности. Ядро определяет уровни прерывания устройствам, число которых зависит от процессора и конфигурации системы. Программные прерывания уровней *IRQL* диспетчерский/*DPC* и *APC* используются ядром для инициирования планирования потоков и асинхронного вмешательства в процесс исполнения потока.

Обработка прерываний. При возникновении прерывания обработчик ловушки сохраняет состояние машины и вызывает диспетчера прерываний. Последний немедленно повышает

IRQL процессора до уровня источника прерывания, чтобы замаскировать прерывания этого и низших уровней на время обработки.

NT использует для поиска обработчика данного прерывания таблицу распределения прерываний (*interrupt dispatch table, IDT*). Индексом таблицы служит *IRQL* источника прерывания, а входы таблицы указывают на процедуры обработки прерываний (рис. 2.15).

После выполнения процедуры обработки прерывания диспетчер прерываний понижает *IRQL* процессора до отметки, на которой он находился перед возникновением прерывания, после чего загружает сохраненное состояние машины. Прерванный поток возобновляет исполнение с точки, в которой произошло прерывание. При снижении ядром *IRQL* могут вскрыться замаскированные прерывания нижних уровней.

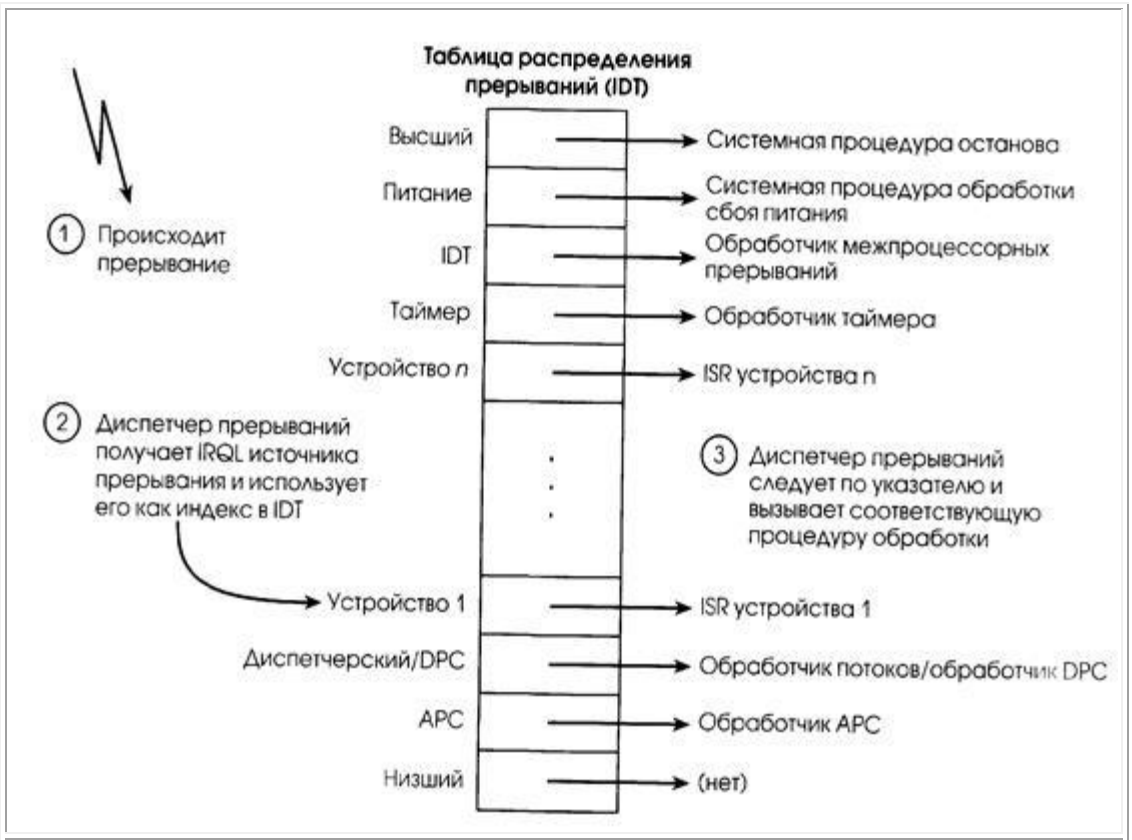

Рис. 2.15. Обслуживание прерывания

Большинство процедур обработки прерываний находится в ядре. Ядро обновляет системное время и выполняет останов системы по сбою питания. Многие прерывания генерируются внешними устройствами, а драйверы сообщают ядру, какую процедуру следует вызвать при возникновении прерываний от соответствующих устройств.

Ядро предоставляет переносимый механизм – объект-прерывание (*interrupt object*), который позволяет драйверам регистрировать *ISR* для своих устройств, содержащий всю информацию, необходимую ядру для того, чтобы связать *ISR* устройства с некоторым уровнем прерываний. Она включает адрес *ISR, IRQL* устройства и вход *IDT* – ядра, с которым должна быть связана *ISR.* Связывание процедуры обработки прерываний с некоторым уровнем прерываний называется подключением объекта-прерывания (*connecting an interrupt object*), а отсоединение от входа *IDT* – отключением объекта-прерывания (*disconnecting an interrupt object*). Эти операции, выполняемые с помощью

вызова функции ядра, позволяют драйверу устройства "включать" *ISR* при его загрузке в систему и снова "отключать" ее, если он выгружается.

Использование объекта-прерывания для регистрации *ISR* позволяет драйверам не работать непосредственно с аппаратурой прерываний, которая различается для разных архитектур процессоров, и избавляет от необходимости знать подробности таблицы распределения прерываний. Это средство ядра помогает создавать переносимые драйверы устройств, так как устраняет необходимость программирования на ассемблере или отражения в драйвере устройства различий между процессорами.

При помощи объекта-прерывания ядро может синхронизировать исполнение *ISR* с другими частями драйвера, возможно – используя общие данные. Более того, объекты-прерывания позволяют ядру легко вызывать более одной *ISR* для данного уровня прерываний. Если несколько драйверов устройств подсоединяют созданные ими объекты-прерывания к одному и тому же входу *IDT*, то при возникновении прерывания на данном уровне диспетчер прерываний вызывает каждую из процедур. Это позволяет ядру легко поддерживать "цепочечные" конфигурации, в которых несколько устройств генерируют прерывания на одном уровне.

Программные прерывания. Хотя большинство прерываний генерируется аппаратурой, ядро *NT* также генерирует программные прерывания в своих целях. Это:

· инициирование планирования потоков;

· обработка истечения интервала таймера;

· асинхронное выполнение процедуры в контексте заданного потока;

· поддержка асинхронного ввода/вывода.

Ниже следуют описания соответствующих задач.

Диспетчерские прерывания. Одно из мест, где ядро использует программные прерывания – в диспетчере потоков. Когда поток не может более выполняться, ядро вызывает диспетчер, чтобы немедленно переключить контекст. Однако иногда ядро обнаруживает необходимость перепланировки, когда оно находится глубоко под несколькими слоями кода. В этой ситуации идеальным решением будет запросить планирование, но отложить его выполнение до того момента, когда ядро завершит текущие действия. Удобный способ реализации – использование программного прерывания.

Для целей синхронизации ядро на время своей работы всегда повышает *IRQL* процессора до уровня диспетчерский/*DPC* или выше, что маскирует программные прерывания (и отключает планирование потоков). Когда ядро определяет, что требуется перепланирование потоков, оно запрашивает прерывание уровня диспетчерский/*DPC*, но, поскольку текущий *IRQL* находится на этом же уровне или выше, то процессор только запоминает данное прерывание. Завершив свою текущую работу, ядро понижает *IRQL* ниже уровня диспетчерский/*DPC*, и диспетчерское прерывание демаскируется.

Прерывания отложенного вызова процедуры (*DPC*). Активизация диспетчера с использованием программного прерывания – это способ отложить переключение потоков

до наступления подходящих условий. *NT* также использует программные прерывания, чтобы отложить выполнение других типов обработки.

Переключение потоков проходит на *IRQL* диспетчерский/*DPC*. Прерывания этого уровня поступают через обработчик ловушки к диспетчеру, который выполняет планирование потоков. "По дороге" ядро обрабатывает также отложенные вызовы процедур (*DPC*). *DPC* – это функция, выполняющая системную задачу, менее важную, чем та, которая выполняется в данный момент. Такие функции называются "отложенными", так как они могут не выполняться сразу. *DPC* выполняются после того, как ядро (или часто система ввода/вывода) закончит выполнение более важной задачи и опустит *IRQL* процессора ниже отметки диспетчерский/*DPC*.

DPC дают ОС возможность генерировать прерывание и исполнять системную функцию в режиме ядра. Ядро использует *DPC* для обработки истечения интервала таймера (и освобождения потоков, ждущих таймеров) и для перепланировки процессора после истечения кванта времени потока. Драйверы устройств используют *DPC* для завершения обработки запросов ввода/вывода.

Представлением *DPC* является объект-*DPC* (*DPC object*) – управляющий объект ядра, который невидим программам пользовательского режима, но видим драйверам устройств и другому системному коду. Самая важная часть информации, хранящейся в объекте-*DPC*, – адрес системной функции, которая вызывается ядром при обработке данного прерывания *DPC*. Ожидающие выполнения процедуры *DPC* хранятся в управляемой ядром очереди, называемой очередью *DP*C. Чтобы запросить *DPC*, системный код обращается к ядру для инициализации объекта-*DPC*, после чего помещает его в эту очередь.

Помещение *DPC* в очередь *DPC* служит указанием ядру запросить программное прерывание уровня диспетчерский/*DPC*. Так как обычно *DPC* помещаются в очередь кодом, исполняющимся на более высоком *IRQL*, то запрошенное прерывание не возникает до тех пор, пока *IRQL* не снизится до уровня *APC* или нижнего уровня.

Так как прерывание *DPC* имеет более низкий приоритет, чем прерывания от устройств, то все ожидающие прерывания от устройств, которые будут демаскированы, обрабатываются до того, как произойдет прерывание *DPC*.

Прерывания асинхронного вызова процедуры (*APC*). Когда ядро помещает в очередь объект-*DPC*, генерируемое в результате этого прерывание *DPC* вмешивается в выполнение любого потока. Для прервания выполнение заданного потока и для выполнения заданной процедуры ядро предоставляет механизм асинхронного вызова процедуры (*APC*). *APC* могут помещаться в очередь как системным кодом, так и кодом пользовательского режима, хотя *APC* режима ядра мощнее. Как и *DPC, APC* выполняется асинхронно при наступлении соответствующих условий. Для пользовательских *APC* такими условиями являются следующие:

· текущим потоком должен быть тот, который переназначен для выполнения данного *APC*;

· *IRQL* процессора должен быть на низшем уровне;

· целевой поток *APC* пользовательского режима должен объявить себя оповещенным;

· *APC* режим ядра, в отличие от *APC* пользовательского режима, не требуют для своего выполнения "разрешения" от целевого потока. Они прерывают поток и выполняют процедуру без его участия или согласия.

Программа помещает в очередь *APC* для некоторого потока, вызывая ядро либо непосредственно (для системного кода), либо косвенно (для кода пользовательского режима). Ядро запрашивает программное прерывание уровня *APC*, и при выполнении всех вышеперечисленных условий целевой поток прерывается и выполняет данный *APC*. Как и *DPC*, *APC* описываются управляющим объектом ядра – объектом-*APC*. *APC*, ждущие своего выполнения, находятся в управляемой ядром очереди *APC* (*APC queue*). В отличие от очереди *DPC*, являющейся общесистемной, очередь *APC* специфична для потока – у каждого потока имеется собственная очередь *APC*. При получении запроса на постановку *APC* в очередь, ядро ставит его в очередь того потока, который выполняет процедуру данного *APC*. *APC* исполняется в контексте заданного потока и на более низком *IRQL*, и к нему не применяются ограничения, налагаемые на *DPC*. Он запрашивает ресурсы (объекты), ждет у описателей объектов, генерирует страничные ошибки и вызывает системные сервисы. Это делает *APC* полезными даже для кода пользовательского режима.

Исполнительная система *NT* использует *APC* режима ядра для выполнения некоторой задачи ОС, которое должно происходить в адресном пространстве (в контексте) определенного потока. Она может использовать *APC* режима ядра, чтобы заставить поток прекратить выполнение системного сервиса. Подсистемы среды применяют *APC* режима ядра, чтобы заставить поток приостановить свое выполнение или завершиться, а также для считывания или установки контекста исполнения пользовательского режима.

Распределение исключений. В отличие от прерываний, которые могут возникать непредсказуемо, исключения – это прямой результат выполнения программы. *Microsoft C* определяет архитектуру программирования, известную как структурная обработка исключений, которая обеспечивает приложениям унифицированную реакцию на исключения. Все исключения, кроме тех, которые достаточно просты и обрабатываются непосредственно обработчиком ловушки, обслуживаются модулем ядра, диспетчерам исключений (*exception dispatcher*) (рис. 2.13). Этот модуль зависит от архитектуры процессора, но написан на *C*. Задача диспетчера исключений состоит в том, чтобы найти обработчик исключений, который может данное исключение "устранить". Ниже приведен список машинно-независимых исключений, определенных ядром:

Нарушение защиты памяти	Целочисленное деление на ноль
Целочисленное переполнение	Переполнение/потеря значимости числа с плавающей точкой
Деление на ноль в арифметике с плавающей точкой	Ненормализованный операнд с плавающей точкой
Отладочная точка останова	Неверное выравнивание данных
Недопустимая машинная команда	Привилегированная машинная команда
Отладочное пошаговое исполнение	Нарушение сторожевой страницы
Ошибка чтения страницы	Исчерпание квоты файла подкачки

Некоторые из этих исключений перехватываются и обрабатываются ядром незаметно для пользовательских программ. Другие исключения ядро обрабатывает, возвращая вызывающей программе код ошибки. Некоторым исключениям позволено "нетронутыми" выходить обратно в пользовательский режим. Для таких исключений подсистема среды

или приложение могут задать при помощи специальных конструкций языка высокого уровня блочные обработчики исключений (*frame based exception handlers*). *Microsoft C* – это первый компилятор *Microsoft*, поддерживающий структурную обработку исключений, однако средства обработки исключений *Windows NT* не зависят от языка программирования.

Термин блочный обозначает связь обработчика исключений с активизацией заданной процедуры. При вызове процедуры в стеке создается стековый фрейм, представляющий активизацию процедуры. Со стековым фреймом может быть связан один или несколько обработчиков исключений, каждый из которых защищает какой-либо блок исходного кода программы. При исключении ядро ищет обработчик, связанный с текущим стековым фреймом. Если его нет, то выполняется поиск обработчика для предыдущего фрейма, пока блочный обработчик исключений не будет найден. Если найти обработчик не удается, то ядро вызывает собственные обработчики исключений по умолчанию.

При возникновении исключения, возбужденого программой или неявно оборудованием, в ядре происходит цепочка событий. Управление передается обработчику ловушки, который создает кадр ловушки (также, как это делается при обработке прерывания). Этот кадр позволяет ОС возобновить выполнение с места возникновения исключения, если последнее было успешно обработано. Кроме того, обработчик ловушки создает запись исключения, которая содержит причину исключения и привходящую информацию.

Если исключение произошло в режиме ядра, диспетчер исключений вызывает процедуру для поиска блочного обработчика, который будет обслуживать данное исключение. Подсистема среды может установить для созданного ею процесса порт отладки и порт исключений. Они используются ядром при обычной обработке исключений, как показано на рис. 2.16. Часто источниками исключений являются отладочные точки останова. Действием диспетчера исключений становится посылка сообщения (через *LPC*) в порт отладки, связанный с процессом, где произошло исключение. Это дает пользователю возможность манипулировать структурами данных и вводить команды отладчика.

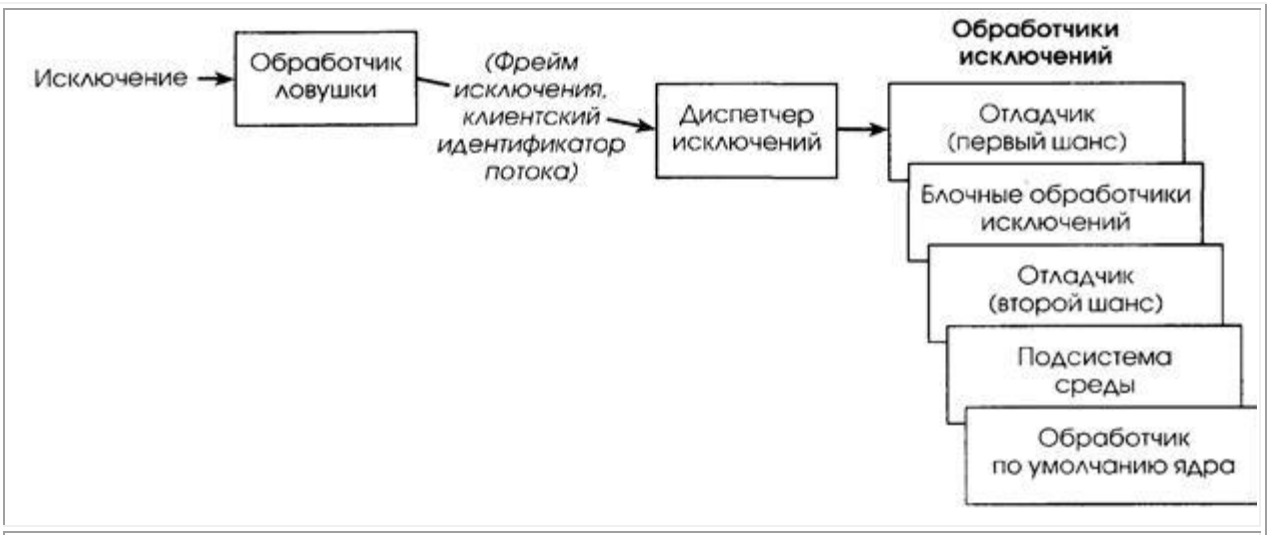

Рис. 2.16. Распределение исключения

Если порт отладки не зарегистрирован или отладчик не обработал исключение, диспетчер исключений переключается в пользовательский режим и вызывает процедуру поиска блочного обработчика исключений. Если обработчик не найден или ни один из них не обработал исключение, то диспетчер исключений переключается обратно в режим ядра и вызывает отладчик, чтобы пользователь мог предпринять действия для отладки. Если

отладчик отсутствует и блочный обработчик не найден, ядро посылает сообщение в порт исключений процесса потока. Порт исключений дает подсистеме среды, которая его просматривает, возможность трансляции исключения *NT* в специфичный для данной среды сигнал или исключение. Если обработка исключения продвинулась до этой точки и подсистема не обслужила исключение, ядро вызывает обработчик исключения по умолчанию, а он просто завершает процесс, поток которого вызвал исключение.

Распределение вызовов системных сервисов. Обработчик ловушки ядра *NT* (рис. 2.13) распределяет прерывания, исключения и вызовы системных сервисов. Вызовы системных сервисов, генерирующие ловушки, – которые в *Windows NT* рассматриваются как исключения – интересны с точки зрения расширяемости системы. Способ реализации системных сервисов ядром позволяет динамически добавлять к ОС новые сервисы в будущих версиях.

Когда поток пользовательского режима вызывает системный сервис, ему вдруг разрешается выполнять привилегированный код ОС. По этой причине процессоры предоставляют команду *syscall* на процессорах *MIPS* и *Intel x*86, если поток пользовательского режима вызывает системный сервис. Аппаратура генерирует ловушку и переключается из пользовательского режима в режим ядра. Когда это происходит, ядро копирует значения параметров сервиса из стека пользовательского режима в стек режима ядра потока (чтобы пользователь никак не мог их изменить) и затем выполняет системный сервис.

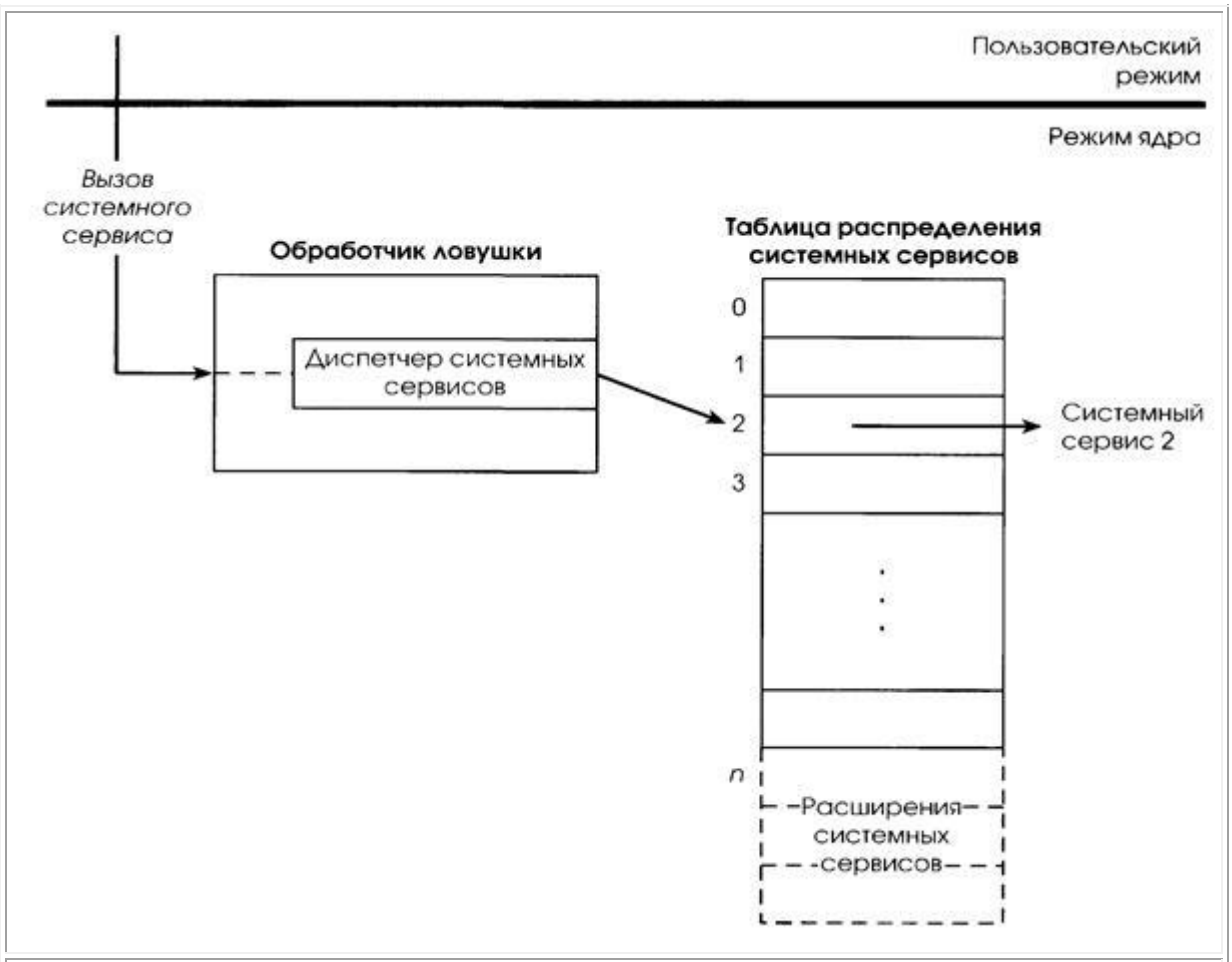

Рис. 2.17. Исключения системных сервисов

Для поиска системных сервисов (рис. 2.17) ядро использует таблицу распределения системных сервисов. Эта таблица похожа на таблицу распределения прерываний, только каждый элемент ее содержит указатель на системный сервис, а не на процедуру обработки.

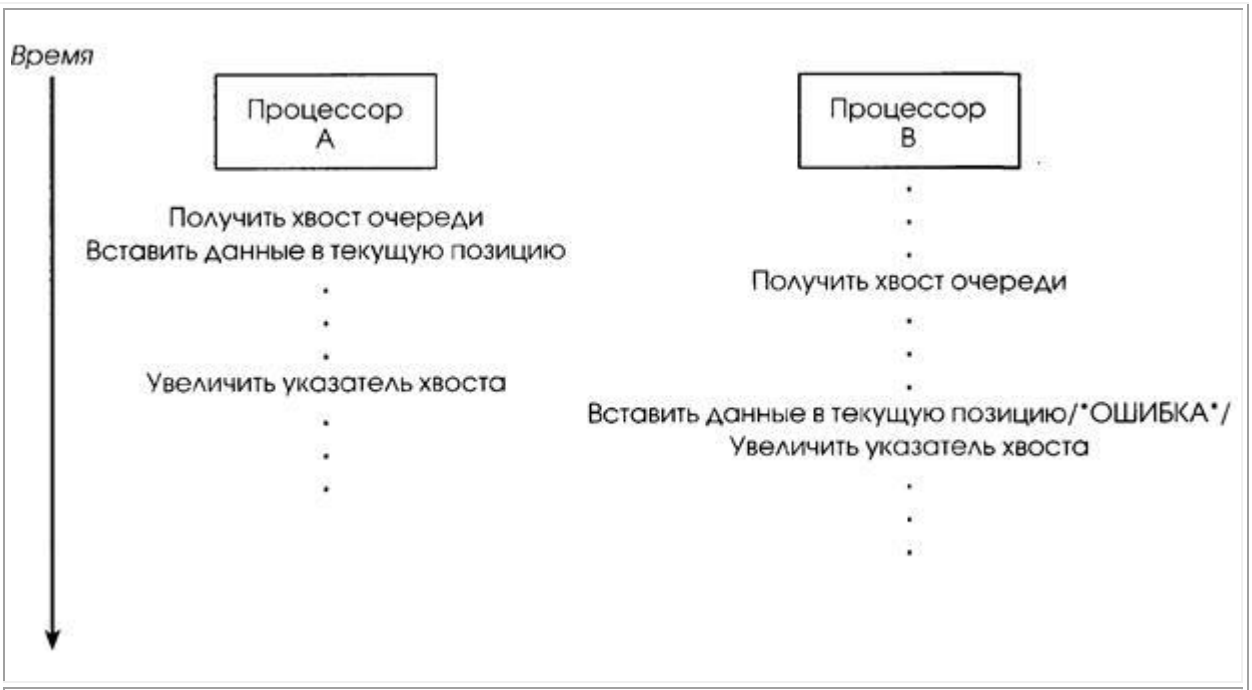

Рис. 2.18. Некорректное совместное использование памяти

Использование таблицы распределения системных сервисов делает базовые сервисы *NT* расширяемыми. Поддержка сервисов добавляется в ядро путем расширения этой таблицы, без изменений в ОС или в приложениях. После создания кода нового сервиса системный администратор запускает утилиту, которая создает таблицу распределения с дополнительным элементом, указывающим на новый системный сервис.

Многопроцессорная синхронизация. Концепция взаимного исключения (*mutual exclusion*) является критической при разработке ОС, т.е. в любой момент времени доступ к данному ресурсу может иметь один и только один поток. Взаимное исключение необходимо, когда ресурс не поддерживает совместный доступ или когда совместное использование дает непредсказуемые результаты. На рис. 2.18 показано, что происходит, когда два потока, выполняющиеся на разных процессорах, выполняют запись в циклическую очередь.

Второй поток получает указатель на конец очереди до того, как первый поток его обновит, второй поток поместит данные на то же место, что и первый – затерев его данные и оставив одно место в очереди пустым. Участки кода, в которых выполняется доступ к ресурсу, не поддерживающему совместное использование, называются критическими секциями (*critical sections*). Чтобы гарантировать корректную работу, в критической секции выполняют не более одного потока. Пока поток записывает данные в файл, обновляет базу данных или модифицирует совместно используемую переменную, никакому другому потоку не разрешен доступ к тому же ресурсу. Код на рис. 2.18 – критическая секция, в которой некорректно, без взаимного исключения, осуществляется доступ к совместно используемым данным.

Взаимное исключение важно для тесно связанных (*tightly-coupled*) ОС с симметричной мультипроцессорной обработкой (*symmetric multiprocessing*), таких как *Windows NT*, где

один и тот же системный код выполняется на нескольких процессорах, совместно используя структуры данных в глобальной памяти. В *Windows NT* предоставление механизмов, которые системный код использует для предотвращения одновременной модификации структуры данных двумя потоками –задача ядра. Ядро обеспечивает примитивы взаимного исключения; оно и другие компоненты исполнительной системы используют их для синхронизации доступа к глобальным структурам данных.

Синхронизация на уровне ядра. На различных стадиях работы ядро должно гарантировать, что в каждый момент времени в критической секции исполняется один и только один процессор. Критические секции ядра – это сегменты кода, модифицирующие глобальные структуры данных, например, базу данных диспетчера ядра или очередь *DPC*. ОС не будет работать нормально, если ядро не гарантирует, что потоки осуществляют доступ к этим данным в режиме взаимного исключения.

В момент изменения ядром глобальной структуры данных может возникнуть прерывание, обработчик которого также изменяет эту структуру. Простые однопроцессорные ОС предотвращают эти случаи, запрещая прерывания на время доступа к глобальным данным, но ядро *NT* прежде чем использовать глобальный ресурс, временно маскирует те прерывания, обработчики которых также используют этот ресурс. Это делается путем повышения *IRQL* процессора до самого высокого уровня. Прерывание уровня диспетчерский/*DPC* вызывает использование диспетчера, который обращается к базе данных диспетчера. Любая другая часть ядра, использующая эту базу данных, повышает *IRQL* до отметки диспетчерский/*DPC*, маскируя прерывания этого уровня перед использованием базы данных. Такая стратегия подходит для однопроцессорной системы, но неадекватна в многопроцессорной конфигурации. Повышение *IROL* на одном процессоре не предотвращает прерываний на других. Ядру необходимо обеспечить взаимоисключающий доступ для нескольких процессоров.

Механизм, используемый ядром для достижения многопроцессорного взаимного исключения, называется спин-блокировкой (*spin lock*), связанный с глобальной структурой данных, например очередью *DPC* (рис. 2.19). Прежде чем войти в критическую секцию, ядро должно получить спин-блокировку, связанную с защищенной очередью *DPC*. Если блокировка не свободна, то ядро пытается получить ее, пока это не увенчается успехом. Спин-блокировка названа так потому, что ядро изолировано и "вращается само по себе", пока не получит блокировку. Спин-блокировки, как и защищаемые ими структуры данных, располагаются в глобальной памяти. Код получения и освобождения спин-блокировки написан на языке ассемблера, чтобы повысить скорость работы и использовать механизм блокировки, предоставляемый данной архитектурой процессора. Во многих архитектурах спин-блокировка реализована при помощи команды "проверить и установить", которая одной операцией проверяет значение переменной блокировки и захватывает блокировку, что предотвращает захват блокировки вторым потоком (в интервале времени между проверкой значения переменной и захватом блокировки первым потоком).

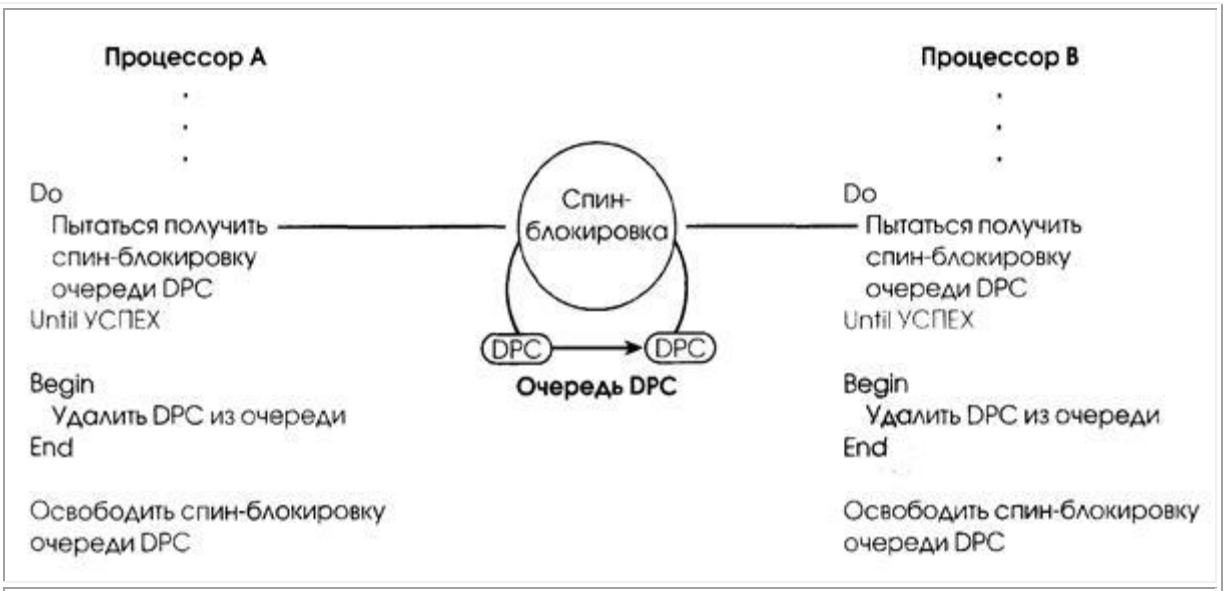

Рис. 2.19. Использование спин-блокировки

Когда поток пытается получить спин-блокировку, вся другая активность на этом процессоре приостанавливается. Поэтому поток, получивший такую блокировку, никогда не вытесняется и имеет возможность продолжать выполнение, чтобы быстрее освободить ее. Ядро использует спин-блокировки с осторожностью, минимизируя количество команд, выполняемых во время обладания ими. Спин-блокировки доступны другим частям исполнительной системы через набор санкций ядра.

Синхронизация на уровне исполнительной системы. Программному обеспечению за пределами ядра необходима синхронизация доступа к глобальным структурам данных в многопроцессорной среде. При помощи функций ядра, исполнительная система создает, получает и освобождает спин-блокировку. Спин-блокировки удовлетворяют потребность исполнительной системы в механизмах синхронизации. Так как ожидание спин-блокировок останавливает процессор, их можно использовать при условиях:

· доступ к защищенному ресурсу должен осуществляться быстро и без сложного взаимодействия с другим кодом;

· критическая секция не может откачиваться из памяти, не может обращаться к нерезидентным данным, не может вызывать внешние процедуры (включая системные сервисы) и не может генерировать прерывания и исключения.

Ядро предоставляет исполнительной системе другие механизмы синхронизации в форме объектов ядра, которые называются диспетчерскими объектами. Поток может синхронизироваться с диспетчерским объектом путем ожидания у его описателя. Это заставляет ядро приостановить поток и изменить его состояние с "исполняющийся" на "ожидающий". Ядро изымает поток из очереди готовности диспетчера.

Поток не может возобновить выполнение, пока ядро не изменит его состояние с "ожидающий" на "готовый". Такое изменение происходит, когда состояние диспетчерского объекта, у описателя которого ждет поток, изменится с "занят" на "свободен". Ядро отвечает за оба типа переходов между состояниями.

Каждый тип диспетчерского объекта предоставляет особый тип синхронизации. Так, объект-мьютекс обеспечивает взаимное исключение; семафор работает как шлюз, через

который может проходить переменное число потоков. События могут использоваться либо для уведомления о том, что было выполнено некоторое действие, либо для реализации взаимного исключения. Пары событий – это средство поддержки ядром быстрого *LPC*, оптимизированной формы передачи сообщений в подсистеме *Win*32. Таймеры "срабатывают" по истечении заданного интервала времени. Поток может ждать завершения другого потока, что бывает полезно для координации действий между взаимодействующими потоками. Все вместе, диспетчерские объекты ядра предоставляют исполнительной системе большую гибкость в синхронизации исполнения.

Синхронизационные объекты, доступные в пользовательском режиме, получают свои возможности синхронизации от диспетчерских объектов ядра. Каждый синхронизационный объект, видимый пользовательскому режиму, инкапсулирует в себе по крайней мере один диспетчерский объект ядра.

Восстановление после сбоя питания.Для сбоя питания в ядре *NT* зарезервировано второе по старшинству приоритета прерывание. Оно уведомляет о проблеме в источнике питания, чтобы система могла выполнить останов корректно. При этом остается время на то, чтобы начать процедуру останова. Если компьютер оборудован резервными батареями для памяти, данные можно восстановить, когда питание будет подано вновь. Выполнявшиеся задачи могут быть перезапущены либо продолжены, в зависимости от их состояния на момент сбоя.

Перезагрузки регистров и возобновления исполнения недостаточно для полного восстановления системы. Так как устройства ввода/вывода работают независимо от остальной ОС, для восстановления после сбоя питания им требуется следующая поддержка со стороны ядра:

· они переинициализируются, когда питание восстановится;

· они должны уметь определять, имел ли место сбой питания.

Эти средства предоставляются двумя управляющими объектами ядра. Объекты – уведомления питания (*power notify objects*) позволяют драйверам устройств зарегистрировать процедуру восстановления, которую ядро будет вызывать при возобновлении питания. Драйвер устройства определяет, что должна делать эта процедура; в общем случае она выполняет повторную инициализацию устройства и перезапуск прерванных операций ввода/вывода. Чтобы зарегистрировать процедуру восстановления после сбоя питания, драйвер создает объект-уведомление питания, вызывает ядро для инициализации объекта указателем на процедуру, после чего снова вызывает ядро для добавления объекта в очередь, контролируемую ядром. При восстановлении питания ядро просматривает эту очередь и вызывает все процедуры по порядку.

Ядро предоставляет еще один управляющий объект, используемый драйверами устройств, объект-состояние питания (*power status object*). Создав такой объект и добавив его в другую очередь ядра, драйвер определяет перед началом операции, которую нельзя прервать (например, запись данных в регистр устройства), не произошел ли сбой питания. Если он произошел, драйвер не выполняет операцию.