

## **Глава 6: Механизмы ядра**

В этой главе обсуждаются различные механизмы, предоставляемые ядром Windows. Некоторые из них полезны для написания драйверов. Другие - это механизмы, которые разработчик драйвера должен понимать, так – как это помогает делать отладку и дает общее понимание устройства системы.

### **В этой главе:**

- Уровень запроса прерывания.
- Отложенные вызовы процедур.
- Асинхронные вызовы процедур.
- Структурированная обработка исключений.
- Системный сбой и отладка.
- Синхронизация потоков.
- Высокая IRQL-синхронизация.
- Рабочие предметы.

### **Уровень запроса прерывания**

В главе 1 мы обсуждали потоки и приоритеты потоков. Эти приоритеты приняты во внимание когда требуется выполнить больше потоков, чем имеется доступных процессоров.

В то же время, аппаратное обеспечение устройства должны уведомлять систему о том, что-то требует внимания. Простой пример Операция ввода/вывода, выполняемая дисководом.

После завершения операции дисковод уведомляет о завершении, запрашивая прерывание. Это прерывание подключено к контроллеру прерываний - аппаратное обеспечение, которое затем отправляет запрос процессору для обработки.

Следующий вопрос, какой поток должен выполнить связанную процедуру обработки прерывания (ISR) ?

Каждое аппаратное прерывание связано с приоритетом, называемым уровнем запроса прерывания (IRQL) (не путать с физической линией прерывания, известной как IRQ), определенной HAL.

У контекста свой IRQL, как и у любого регистра.

IRQL могут быть или не быть реализованы аппаратным обеспечением процессора, но это принципиально неважно.

IRQL должен рассматриваться как любой другой процессорный регистр.

Основное правило заключается в том, что процессор выполняет код с самым высоким IRQL.

Например, если IRQL процессора в какой-то момент равен нулю, и появляется прерывание с соответствующим IRQL 5, процессор сохраняет свое состояние (контекст) в стеке ядра текущего потока, поднимает IRQL до 5 и затем выполняет ISR, связанный с прерыванием.

После завершения ISR IRQL опустится до своего предыдущего уровня, возобновляя предыдущий выполненный код, как если бы прерывание не существовало.

Если с другой стороны, IRQL нового прерывания выше 5, процессор снова сохранит свое состояние, повысит IRQL на новый уровень, выполнит второй ISR, связанный со вторым прерыванием, и, когда он завершен, вернется к IRQL 5, восстановит его состояние и продолжит выполнение исходного ISR.

По сути, повышение IRQL временно блокирует код с равным или меньшим значением IRQL.

На следующем рисунке показано работа диспетчеризации прерываний.

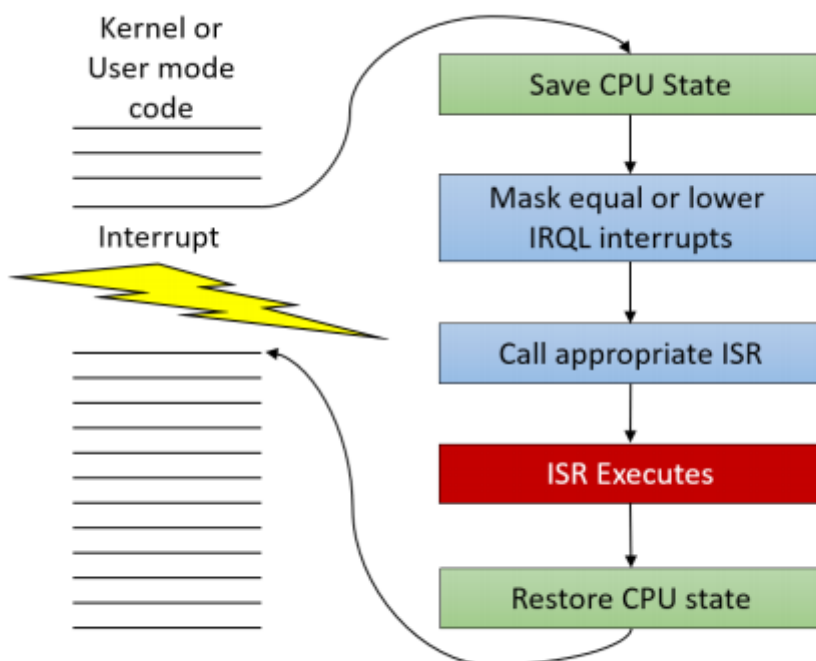


Figure 6-1: Basic interrupt dispatching

На следующем рисунке показано как выглядит вложенность прерываний.

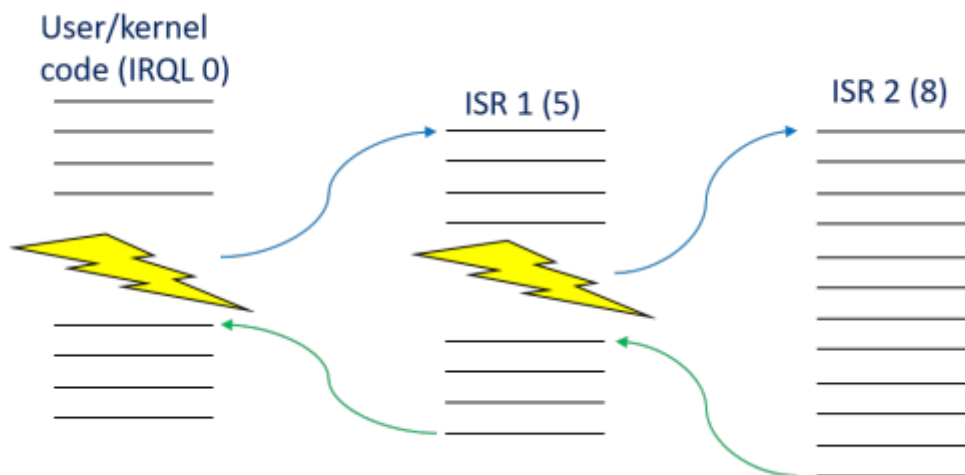


Figure 6-2: Nested interrupts

Windows не имеет специального потока для обработки прерываний, они обрабатываются тем потоком, который был запущен.

Как мы скоро обнаружим, переключение контекста невозможно, когда IRQL процессора имеет значение 2 или выше, поэтому другой поток не сможет проникнуть во время выполнения этих ISR.

Когда выполняется код пользовательского режима, IRQL всегда равен нулю. Это одна из причин, почему термин IRQL не упоминается ни в одной документации пользовательского режима - он всегда равен нулю и не может быть изменен.

Часто код режима ядра также работает с нулевым IRQL. В режиме ядра возможно поднять IRQL.

#### **Важные IRQL описаны ниже:**

- `PASSIVE_LEVEL` в WDK (0) - это «нормальный» IRQL для процессора. Код режима пользователя всегда работает на этом уровне. Планирование потоков работает нормально, как описано в главе 1.
- `APC_LEVEL` (1) - используется для специальных APC ядра (будут обсуждаться в этой главе, в разделе асинхронные вызовы процедур).

Планирование потоков работает нормально.

- `DISPATCH_LEVEL` (2) - здесь все радикально меняется. Планировщик не может проснуться. Доступ к выгружаемой памяти не разрешен - такой доступ вызывает сбой системы. Поскольку планировщик не может вмешиваться, ожидание на объектах ядра не допускается.

- IRQL устройства - диапазон уровней, используемых для аппаратных прерываний (от 3 до 11 на x64/ARM/ARM64, от 3 до 26 на x86). Все правила из IRQL 2 применяются и здесь.

- Самый высокий уровень (HIGH\_LEVEL) - это самый высокий IRQL, маскирующий все прерывания. Используется некоторыми API, занимающиеся манипулированием связанными со списками. Фактические значения 15 (x64 ARM ARM64) и 31 (x86).

Когда IRQL процессора повышается до 2 или выше (по любой причине), на исполняемый код накладываются следующие ограничения:

- Доступ к памяти вне физической памяти является фатальным и вызывает сбой системы. Это означает доступ к данным из невыгружаемого пула всегда безопасен, тогда как доступ к данным из выгружаемого пула или от предоставленных пользователем буферов не является безопасным и его следует избегать.
- Ожидание любого объекта ядра диспетчера (например, мьютекса или события) вызывает сбой системы, если только тайм-аут ожидания равен нулю, что все еще разрешено. (мы обсудим объект диспетчера в этой главе в разделе «Синхронизация».)

Эти ограничения связаны с тем, что планировщик «работает» на IRQL 2; так что если IRQL процессора уже 2 или выше, планировщик не может проснуться на этом процессоре, поэтому переключение контекста не может возникнуть.

Только прерывания более высокого уровня могут временно перенаправить код в связанный ISR, но это все тот же поток - без переключения контекста.

Контекст потока сохраняется, выполняется ISR и состояние потока возобновляется.

## 2)Повышение и понижение IRQL

Как уже говорилось ранее, в пользовательском режиме понятие IRQL не упоминается, и нет никакого способа изменить это.

В режиме ядра IRQL можно повысить с помощью функции KeRaiseIrql и понизить вернуться с KeLowerIrql.

Вот фрагмент кода, который поднимает IRQL до DISPATCH\_LEVEL (2), и затем опускает его обратно после выполнения некоторых инструкций на этом IRQL.

```
// assuming current IRQL <= DISPATCH_LEVEL
KIRQL oldIrql; // typedefed as UCHAR
KeRaiseIrql(DISPATCH_LEVEL, &oldIrql);
NT_ASSERT(KeGetCurrentIrql() == DISPATCH_LEVEL);
// do work at IRQL DISPATCH_LEVEL
KeLowerIrql(oldIrql);
```

Если вы поднимаете IRQL, убедитесь, что вы опускаете его в той же функции. Слишком опасно возвращаться из функции с более высоким IRQL.

Также убедитесь, что KeRaiseIrql фактически повышает IRQL, а KeLowerIrql фактически понижает его, в противном случае будет сбой системы.

## Приоритеты потоков по сравнению с IRQL

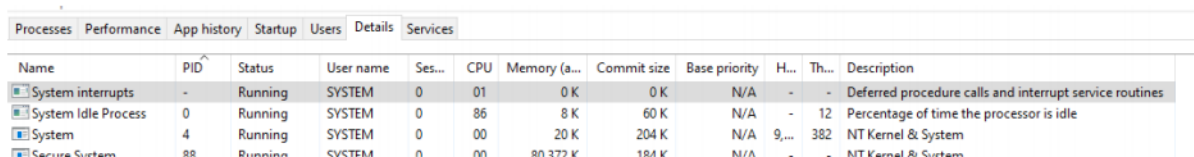
IRQL - это атрибут процессора. Приоритет является атрибутом потока. Приоритеты потоков имеют только значение при  $IRQL < 2$ .

Как только исполняющий поток повысил IRQL до 2 или выше, его приоритет больше ничего не значит - теоретически он имеет бесконечный квант - он будет продолжать выполнение пока IRQL не будет ниже 2.

Естественно, тратить много времени на  $IRQL > 2$  не очень хорошая вещь. Это только одна из причин, по которой существуют строгие ограничения на то, что может делать исполняемый код на этом уровне.

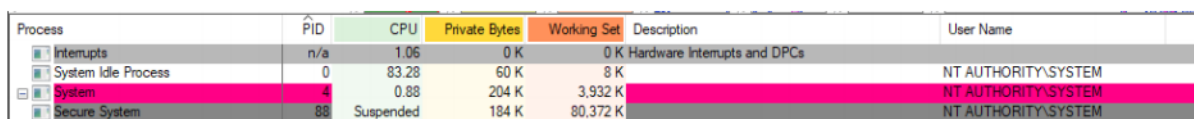
Диспетчер задач показывает количество процессорного времени, проведенного в IRQL 2 или выше, используя псевдопроцесс, который называется системным прерыванием, в Process Explorer называет это прерываниями.

Следующий рисунок показывает снимок экрана с Диспетчер задач и рисунок 6-4 показывают ту же информацию в Process Explorer.



Name	PID	Status	User name	Ses...	CPU	Memory (a...	Commit size	Base priority	H...	Th...	Description
System interrupts	-	Running	SYSTEM	0	01	0 K	0 K	N/A	-	-	Deferred procedure calls and interrupt service routines
System Idle Process	0	Running	SYSTEM	0	86	8 K	60 K	N/A	-	12	Percentage of time the processor is idle
System	4	Running	SYSTEM	0	00	20 K	204 K	N/A	9...	382	NT Kernel & System
Secure System	88	Running	SYSTEM	0	00	80,372 K	184 K	N/A	-	-	NT Kernel & System

Figure 6-3: IRQL 2+ CPU time in Task Manager



Process	PID	CPU	Private Bytes	Working Set	Description	User Name
Interrupts	n/a	1.06	0 K	0 K	Hardware Interrupts and DPCs	
System Idle Process	0	83.28	60 K	8 K		NT AUTHORITY\SYSTEM
System	4	0.88	204 K	3,932 K		NT AUTHORITY\SYSTEM
Secure System	88	Suspended	184 K	80,372 K		NT AUTHORITY\SYSTEM

Figure 6-4: IRQL 2+ CPU time in Process Explorer

## Отложенный вызов процедур

Рисунок 6-5 показывает типичную последовательность событий, когда клиент вызывает некоторую операцию ввода-вывода.

В этом рисунке поток пользовательского режима открывает дескриптор файла и выполняет операцию чтения с использованием ReadFile.

Поскольку поток может сделать асинхронный вызов, он почти сразу же получает управление и может делать другую работу. Драйвер, получающий этот запрос, вызывает драйвер файловой системы (например, NTFS), который может вызывать другие драйверы под ним, пока запрос не достигнет драйвера диска, который инициирует операции на реальном оборудовании диска.

На этом этапе не нужно выполнять код, так как аппаратное обеспечение «Делает свое дело».

Когда оборудование завершает операцию чтения, оно выдает прерывание. Это вызывает подпрограмму службы, связанной с прерыванием.

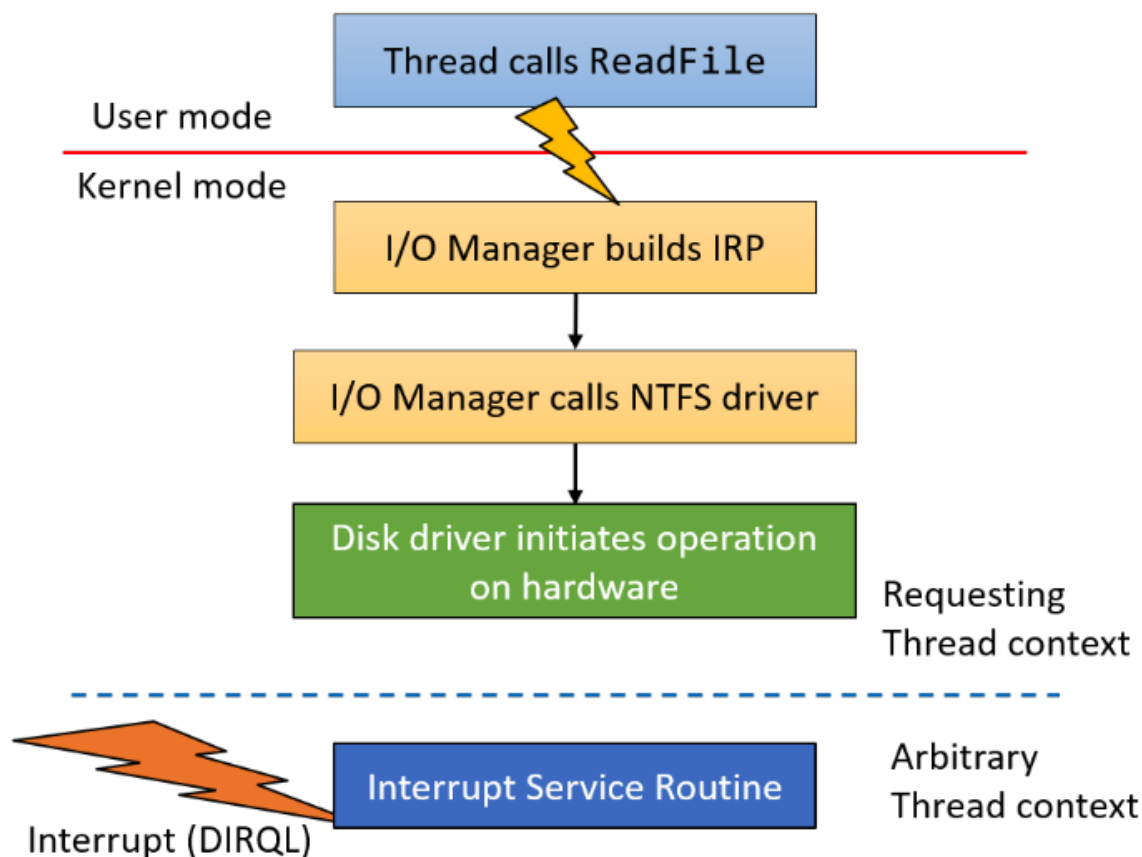


Figure 6-5: Typical I/O request processing (part 1)

Как мы видели в главе 4, выполнение запроса выполняется путем вызова `IoCompleteRequest`. Проблема в том, что в документации говорится, что эту функцию можно вызывать только при `IRQL <= DISPATCH_LEVEL(2)`. Это означает, что ISR не может вызвать `IoCompleteRequest`, или это приведет к сбою системы.

Так что же ISR делать?

Вы можете задаться вопросом, почему существует такое ограничение. Одна из причин связана с работой `IoCompleteRequest`.

Мы обсудим это более подробно в следующей главе, но суть в том, что эта функция относительно дорогая.

Это будет означать, что выполнение ISR займет значительно больше времени, и, поскольку он выполняется в высокий IRQL, он будет маскировать другие прерывания в течение более длительного периода времени.

Механизм, который позволяет ISR вызывать IoCompleteRequest (и другие функции с аналогичными ограничения) как можно скорее использует отложенный вызов процедуры (DPC).

DPC является объектом инкапсуляция функции, которая должна вызываться в IRQL DISPATCH\_LEVEL.

Вы можете спросить, почему ISR просто не понижает текущий IRQL до DISPATCH\_LEVEL, затем не вызывает IoCompleteRequest и затем возвращает IRQL к его первоначальному значению ?

Это может привести в тупик. Мы обсудим причину этого позже в этой главе в разделе «Спин-блокировки».

Драйвер, который зарегистрировал ISR, готовит DPC заранее, выделяя структуру KDPC из невыгружаемого пула и инициализации его с помощью функции обратного вызова с использованием KeInitializeDpc.

Затем, когда вызывается ISR, непосредственно перед выходом из функции ISR запрашивает DPC для выполнения как можно скорее, поставив его в очередь, используя KeInsertQueueDpc.

Когда функция DPC выполняется, она вызывает IoCompleteRequest. Таким образом, DPC служит компромиссом - он работает на IRQL DISPATCH\_LEVEL.

Это означает, что планирование не может выполняться, нет доступа к памяти и т. д., но оно недостаточно высоко, чтобы предотвратить аппаратные прерывания, которые поступают и обслуживаются на одном процессоре.

Когда ISR возвращается, прежде чем IRQL может упасть до нуля выполняется проверка, чтобы увидеть, существуют ли ЦП в очереди процессора.

Если есть, то процессор сбрасывается до IRQL DISPATCH\_LEVEL (2), а затем обрабатывает DPC в очереди в первую очередь.

В порядке первого выхода (FIFO), вызывая соответствующие функции, пока очередь не станет пустой. Только тогда IRQL процессора упадет до нуля и возобновит выполнение исходного кода, который был нарушен во время исполнения прерывания.

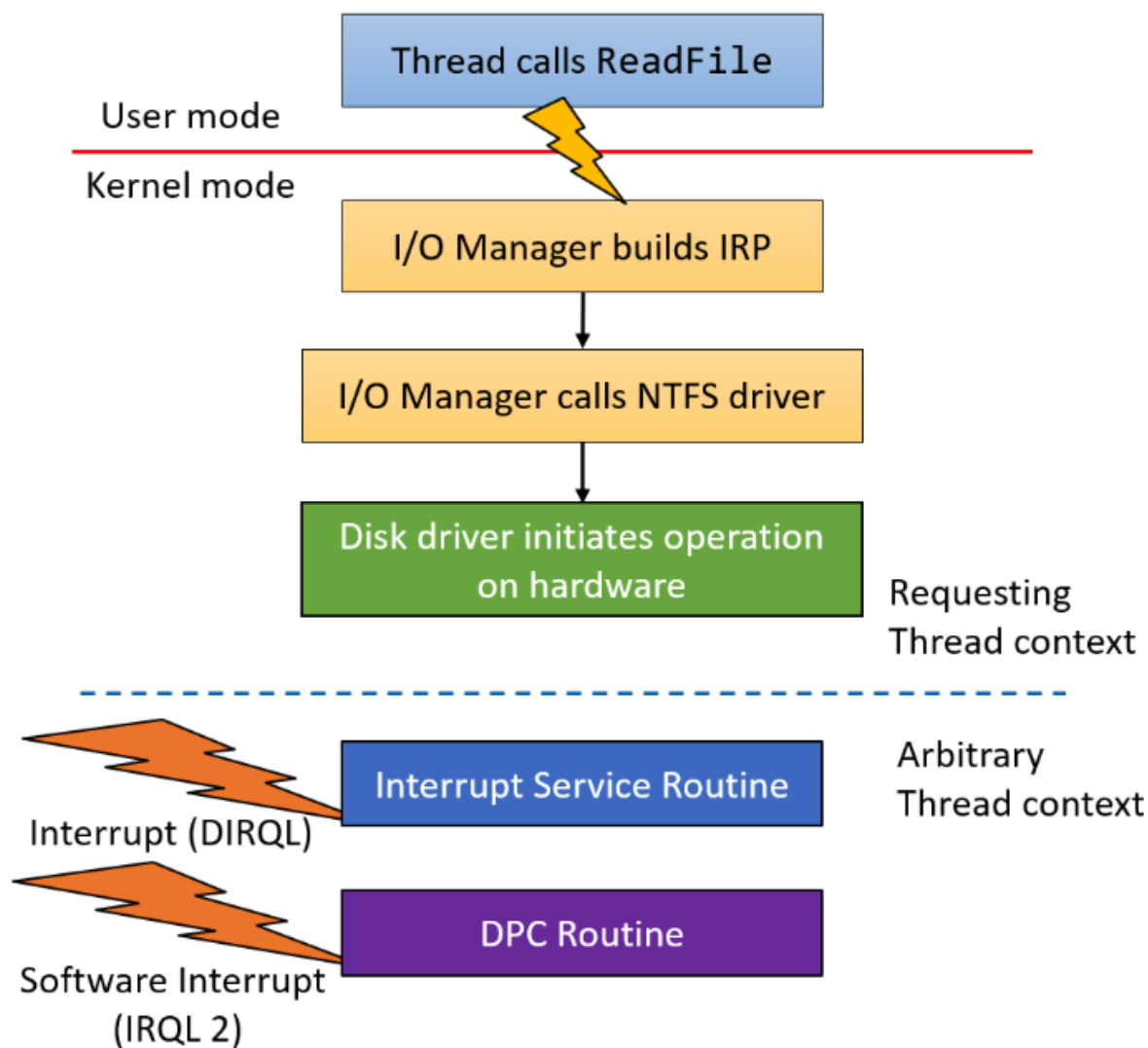


Figure 6-6: Typical I/O request processing (part 2)

### Использование DPC с таймером

DPC были изначально созданы для использования ISR. Однако в ядре есть и другие механизмы которые используют DPC.

Одним из таких применений является таймер ядра. Таймер ядра, представленный структурой KTIMER, позволяет установить таймера для ожидания некоторого времени в будущем, на основе относительного интервала или абсолютного времени.

Этот таймер является объектом диспетчера, поэтому его можно ожидать с помощью KeWaitForSingleObject (Будет обсуждено далее в этой главе в разделе «Синхронизация»).

Хотя ожидание возможно, но это неудобно для таймера. Более простой подход состоит в том, чтобы вызвать некоторый обратный вызов, когда таймер истекает.

Это именно то, что таймер ядра обеспечивает использование DPC в качестве обратного вызова.

В следующем фрагменте кода показано, как настроить таймер и связать его с DPC.



Когда таймер истекает, DPC вставляется в очередь DPC CPU и, таким образом, выполняется как можно скорее.

Метод с помощью таймера, является более мощным, чем нулевой обратный вызов на основе IRQL, поскольку он гарантированно выполняется до любого кода режима пользователя (и большинство кодов режима ядра).

```
KTIMER Timer;
```

```
KDPC TimerDpc;
```

```
void InitializeAndStartTimer(ULONG msec) {  
    KeInitializeTimer(&Timer);  
    KeInitializeDpc(&TimerDpc,  
                   OnTimerExpired,  
                   // callback function  
                   nullptr);  
    // passed to callback as "context"  
    // relative interval is in 100nsec units (and must be negative)  
    // convert to msec by multiplying by 10000  
    LARGE_INTEGER interval;  
    interval.QuadPart = -10000LL * msec;  
    KeSetTimer(&Timer, interval, &TimerDpc);  
}  
  
void OnTimerExpired(KDPC* Dpc, PVOID context, PVOID, PVOID) {  
    UNREFERENCED_PARAMETER(Dpc);  
    UNREFERENCED_PARAMETER(context);  
    NT_ASSERT(KeGetCurrentIrql() == DISPATCH_LEVEL);  
    // handle timer expiration  
}
```

### Асинхронные вызовы процедур

В предыдущем разделе мы видели, что DPC - это объекты, инкапсулирующие функцию, вызываемую в IRQL DISPATCH\_LEVEL.

Вызывающий поток не имеет значения, поскольку речь идет о DPC.

Асинхронные вызовы процедур (APC) также являются структурами данных, которые инкапсулируют функцию. Но в отличие от DPC, APC ориентирован на определенный поток, поэтому только этот поток может выполнить функцию.

Это означает, что с каждым потоком связана очередь APC.

Существует три типа APC:

- APC пользовательского режима - они выполняются в пользовательском режиме на IRQL PASSIVE\_LEVEL, только когда поток переходит в состояние готовности.

Обычно это достигается путем вызова API, такого как SleepEx, WaitForSingleObjectEx, WaitForMultipleObjectsEx и аналогичные API.

Последний аргумент для этих функций может быть установлено значение TRUE, чтобы перевести поток в состояние уведомления. В этом состоянии он смотрит на очередь APC, и APC теперь выполняются до тех пор, пока очередь не станет пустой.

- Обычные APC режима ядра - они выполняются в режиме ядра на IRQL PASSIVE\_LEVEL.
- Специальные APC ядра - они выполняются в режиме ядра на IRQL APC\_LEVEL (1).

Эти APC используются системой ввода/вывода, завершают операции ввода/вывода. Общий сценарий этого будет обсуждаться в следующей главе.

API APC недокументирован в режиме ядра, поэтому драйверы обычно не используют APC напрямую.

Пользовательский режим может использовать APC, вызывая определенные API. Например, вызвав ReadFileEx или WriteFileEx запускают асинхронную операцию ввода-вывода.

По завершении APC пользовательского режима присоединяется к вызывающему потоку.

Этот APC будет выполняться, когда поток переходит в состояние оповещения, как описано ранее.

Еще одна полезная функция в пользовательском режиме позволяющая явно генерировать APC является QueueUserAPC.

## **Критические регионы и защищенные регионы**

Критическая область препятствует выполнению пользовательского режима и обычных APC ядра (специальные APC ядра все еще можно выполнить).

Поток входит в критическую область с KeEnterCriticalRegion и покидает его с KeLeaveCriticalRegion.

Некоторые функции в ядре требуют быть внутри критической области, особенно при работе с исполнительными ресурсами (см. раздел «Исполнительные ресурсы» далее в этой глава).

Охраняемая область препятствует выполнению всех APC. Вызовите KeEnterGuardedRegion, чтобы ввести охраняемый регион и KeLeaveGuardedRegion что-бы покинуть его.

Рекурсивные вызовы KeEnterGuardedRegion должно совпадать с одинаковым количеством вызовов KeLeaveGuardedRegion.

## Структура обработчика исключений

Исключением является событие, которое происходит из-за определенной инструкции, которая сделала что-то, что вызвало ошибку процессора. Исключения в чем-то похожи на прерывания, главное отличие это то-что исключения является синхронным и технически воспроизводимыми при одних и тех-же условиях, тогда как прерывание асинхронны и может появиться в любое время.

Примеры исключений включают деление на ноль, точка останова, ошибка страницы, переполнение стека и неверная инструкция.

Если возникает исключение, ядро ловит это и позволяет коду обрабатывать исключение, если это возможно.

Этот механизм называется структурированной обработкой исключений (SEH) и доступен для кода режима пользователя.

Обработчики исключений ядра вызываются на основе той же таблицы обработки прерываний (IDT).

Используя отладчик ядра, команда !id показывает все эти сопоставления. Векторы прерываний с низким номером фактически являются обработчиками исключений.

Вот пример вывода из этой команды:

```
lkd> !idt

Dumping IDT: fffff8011d941000

00: fffff8011dd6c100 nt!KiDivideErrorFaultShadow
01: fffff8011dd6c180 nt!KiDebugTrapOrFaultShadow      Stack = 0xFFFFF8011D9459D0
02: fffff8011dd6c200 nt!KiNmiInterruptShadow          Stack = 0xFFFFF8011D9457D0
03: fffff8011dd6c280 nt!KiBreakpointTrapShadow
04: fffff8011dd6c300 nt!KiOverflowTrapShadow
05: fffff8011dd6c380 nt!KiBoundFaultShadow
06: fffff8011dd6c400 nt!KiInvalidOpcodeFaultShadow
07: fffff8011dd6c480 nt!KiNpxNotAvailableFaultShadow
08: fffff8011dd6c500 nt!KiDoubleFaultAbortShadow      Stack = 0xFFFFF8011D9453D0
09: fffff8011dd6c580 nt!KiNpxSegmentOverrunAbortShadow
0a: fffff8011dd6c600 nt!KiInvalidTssFaultShadow
0b: fffff8011dd6c680 nt!KiSegmentNotPresentFaultShadow
0c: fffff8011dd6c700 nt!KiStackFaultShadow
0d: fffff8011dd6c780 nt!KiGeneralProtectionFaultShadow
0e: fffff8011dd6c800 nt!KiPageFaultShadow
10: fffff8011dd6c880 nt!KiFloatingErrorFaultShadow
11: fffff8011dd6c900 nt!KiAlignmentFaultShadow

(truncated)
```

Обратите внимание на названия функций - большинство из них очень наглядны.

Некоторые распространенные примеры исключений включают в себя:

- Деление на ноль (0)
- Точка останова (3) - ядро выполняет это прозрачно, передавая управление подключенному отладчику (если есть).
- Неверный код операции (6) - эта ошибка вызывается процессором, если он встречает неизвестную инструкцию.
- Ошибка страницы (14) - эта ошибка вызывается ЦП, если страница не находится в физической памяти.

### **Нарушение исключения.**

Как только возникает исключение, ядро ищет функцию, для обработки исключения.

Если обработчик исключения не найден, ядро будет искать стек вызовов, пока такой обработчик не будет найден.

Если стек вызовов исчерпан, система рухнет.

Как драйвер может обрабатывать эти типы исключений? Microsoft добавила четыре ключевых слова к языку Си чтобы позволить разработчикам легко обрабатывать такие исключения.

Добавленные ключевые слова и краткое описание:

`__try` - Запускает блок кода, где могут возникать исключения.

`__except` - Указывает, обрабатывается ли исключение, и предоставляет код обработки, если это так.

`__finally` - Не связано с исключениями напрямую. Предоставляет код, который гарантированно выполняется независимо от возникло-ли исключение.

`__leave` - Предоставляет оптимизированный механизм для перехода к блоку `__finally` внутри `__try` блока.

### **Использование `__try` / `__except`**

В главе 4 мы реализовали драйвер, который обращается к буферу пользовательского режима для получения данных, необходимых для операций в драйвере.

Мы использовали прямой указатель на буфер пользователя.

Однако это не безопасно. Например, код режима пользователя (скажем, из другого потока) может освободить буфер непосредственно перед тем, как драйвер получит к нему доступ.

В таком случае драйвер вызовет сбой системы, в основном из-за ошибки пользователя (или злого умысла). Поскольку пользовательским данным никогда нельзя доверять, такой доступ должен быть обернут в блок `__try/__except`, чтобы убедиться, что плохой буфер не приводит к сбою драйвера.

Вот важная часть пересмотренного обработчика IRP\_MJ\_DEVICE\_CONTROL, использующего обработчик исключения:

```
case IOCTL_PRIORITY_BOOSTER_SET_PRIORITY:
{
    if (stack->Parameters.DeviceIoControl.InputBufferLength <
        sizeof(ThreadData)) {
        status = STATUS_BUFFER_TOO_SMALL;
        break;
    }

    auto data = (ThreadData*)stack-
>Parameters.DeviceIoControl.Type3InputBuffer;

    if (data == nullptr) {
        status = STATUS_INVALID_PARAMETER;
        break;
    }

    __try {
        if (data->Priority < 1 || data->Priority > 31) {
            status = STATUS_INVALID_PARAMETER;
            break;
        }

        PETHREAD Thread;

        status = PsLookupThreadByThreadId(ULONGToHandle(data->ThreadId),
&Thread);

        if (!NT_SUCCESS(status))
            break;

        KeSetPriorityThread((PKTHREAD)Thread, data->Priority);
        ObDereferenceObject(Thread);

        KdPrint(("Thread Priority change for %d to %d succeeded!\n",
            data->ThreadId, data->Priority));
    }

    __except (EXCEPTION_EXECUTE_HANDLER) {
        // something wrong with the buffer

        status = STATUS_ACCESS_VIOLATION;
    }
}
```

```

_____]
_____break;
}

```

Размещение EXCEPTION\_EXECUTE\_HANDLER в \_\_except говорит о том, что любое исключение должно быть обработано. Мы можем быть более избирательным, вызывая GetExceptionCode и просматривая фактическое исключение. Если мы не ожидаем этого, мы можем сказать ядру продолжить поиск обработчиков в стеке вызовов:

```

_____]except (GetExceptionCode() == STATUS_ACCESS_VIOLATION
_____? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
_____// handle exception
}

```

Например, нарушение доступа может быть обнаружено только в том случае, если нарушенный адрес принадлежит пользовательскому пространству.. Если он находится в пространстве ядра, он не будет перехвачен и все равно вызовет сбой системы.

Механизм SEH также может использоваться драйверами (и кодом пользовательского режима) для создания пользовательских исключений.

Ядро предоставляет обобщенную функцию ExRaiseStatus для вызова любых исключений и некоторых конкретных функции, такие как ExRaiseAccessViolation.

### Использование \_\_try/\_\_finally

Использование блоков \_\_try и \_\_finally не имеет прямого отношения к исключениям. Это похоже на концепцию ключевого слова finally, на каком-то высоко уровне языке (например, Java, C #). Вот простой пример, чтобы показать проблему:

```

void foo() {
    void* p = ExAllocatePool(PagedPool, 1024);
    // do work with p
    ExFreePool(p);
}

```

Приведенный выше код кажется достаточно безвредным. Однако, есть несколько проблем с этим:

- Если между выделением памяти и обработкой будет исключение, обработчик в вызывающей стороне не освободит память.
- если оператор return используется в некотором условном выражении между выделением и освобождением, буфер не будет освобожден.

Используя комбинацию `__try/__finally`, мы можем убедиться, что буфер будет освобожден всегда:

```
void foo() {  
    void* p = ExAllocatePool(PagedPool, 1024);  
    __try {  
        // do work with p  
    }  
    __finally {  
        ExFreePool(p);  
    }  
}
```

С указанным выше кодом, даже если в теле `__try` появляются операторы возврата, код будет вызван перед фактическим возвратом из функции.

Если возникает какое-то исключение, блок `__finally` запускается первым до того, как ядро выполнит поиск в стеке вызовов обработчиков.

`__try/__finally` полезен не только для выделения памяти, но и для других ресурсов.

Один общий пример - при синхронизации, потоки обращаются к некоторым общим данным. Вот пример получения и освобождения быстрого мьютекса (мьютекс и другие примитивы синхронизации описаны далее в этой главе):

```
FAST_MUTEX MyMutex;
```

```
void foo() {  
    ExAcquireFastMutex(&MyMutex);  
    __try {  
        // do work while the fast mutex is held  
    }  
    __finally {  
        ExReleaseFastMutex(&MyMutex);  
    }  
}
```

### Использование C++ RAII вместо `__try / __finally`

Хотя предыдущие примеры с `__try/__finally` работают, они не очень удобны.

Используя C++, мы можем создавать оболочки RAII, которые делают правильные вещи без необходимости использовать \_\_try/\_\_finally.

В C++ нет ключевого слова finally, такого как C# или Java, но оно не требуется - есть-же деструкторы.

Вот очень простой пример, который управляет распределением буфера с помощью класса RAII:

```
template<typename T = void>
struct kunique_ptr {
    kunique_ptr(T* p = nullptr) : _p(p) {}
    ~kunique_ptr() {
        if (_p)
            ExFreePool(_p);
    }
    T* operator->() const {
        return _p;
    }
    T& operator*() const {
        return *_p;
    }
private:
    T* _p;
};
```

Класс использует шаблоны, позволяющие легко работать с любым типом данных. Пример использования следующий:

```
struct MyData {
    ULONG Data1;
    HANDLE Data2;
};

void foo() {
    // take charge of the allocation
    kunique_ptr<MyData> data((MyData*)ExAllocatePool(PagedPool,
sizeof(MyData)));
```



```

    // use the pointer

    data->Data1 = 10;

    // when the object goes out of scope, the destructor frees the buffer
}

```

Если вы обычно не используете C++ в качестве основного языка программирования, вы можете подумать, что приведенный выше код запутанный. Вы можете продолжить работу с `__try/__finally`, но я рекомендую познакомиться с этим типом кода. В любом случае, даже если вы боретесь с реализацией `kunique_ptr` выше, вы все еще можете использовать его без необходимости понимать каждую мелочь.

Представленный тип `kunique_ptr` - это минимум. Вы также должны удалить конструктор копирования и т.д.

Вот более полная реализация:

```

template<typename T = void>
struct kunique_ptr {
    kunique_ptr(T* p = nullptr) : _p(p) {}

    // remove copy ctor and copy = (single owner)

    kunique_ptr(const kunique_ptr&) = delete;
    kunique_ptr& operator=(const kunique_ptr&) = delete;

    // allow ownership transfer

    kunique_ptr(kunique_ptr&& other) : _p(other._p) {
        other._p = nullptr;
    }

    kunique_ptr& operator=(kunique_ptr&& other) {
        if (&other != this) {
            Release();

            _p = other._p;
            other._p = nullptr;
        }

        return *this;
    }

    ~kunique_ptr() {
        Release();
    }
}

```

```

_____]
_____] operator bool() const {
_____] return _p != nullptr;
_____]
_____] T* operator->() const {
_____] return _p;
_____]
_____] T& operator*() const {
_____] return *_p;
_____]
_____]
_____] void Release() {
_____] if (_p)
_____] ExFreePool(_p);
_____]
_____]
private:
_____] T* _p;
_____]
};

```

Мы создадим другие оболочки RAII для примитивов синхронизации позже в этой главе.

### Системный сбой и отладка

Как мы уже знаем, если в режиме ядра возникает необработанное исключение, система дает сбой, обычно с «синим экраном смерти» (BSOD).

В этом разделе мы обсудим, что происходит при сбое системы и как с этим бороться.

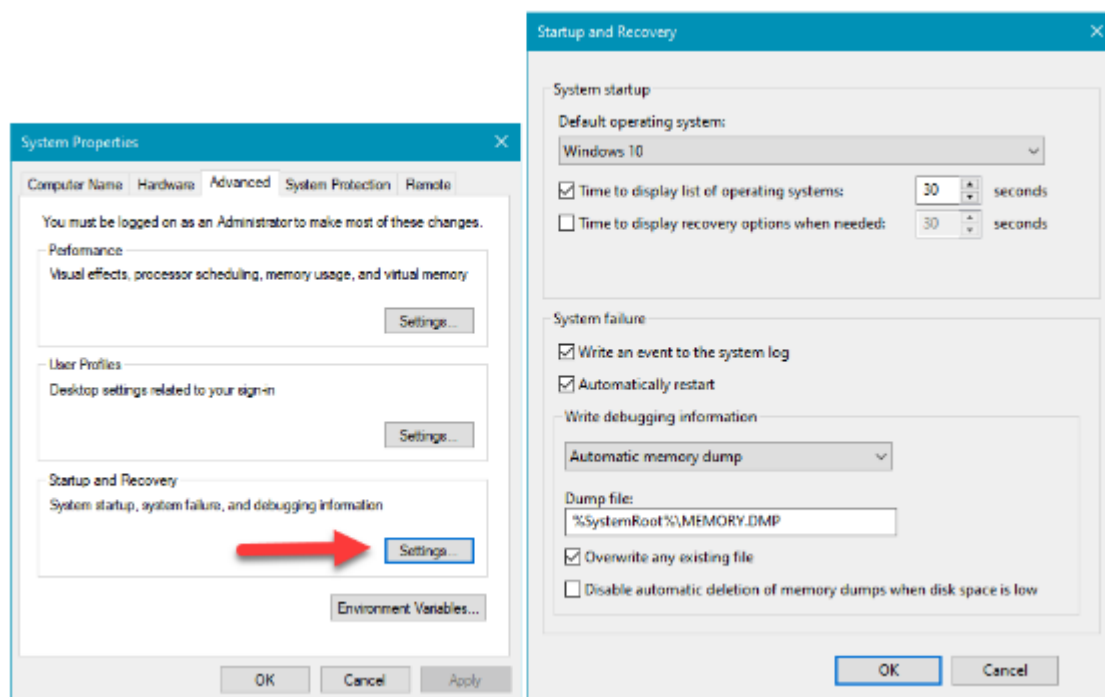
Если код ядра, которому предполагается доверять, сделал что-то плохое, остановить все, это вероятно, самый безопасный подход, поскольку, возможно, если позволить коду продолжать перемещаться, это может привести к повреждению некоторых важных файлов или ключей реестра.

*В последних версиях Windows 10 есть несколько альтернативных цветов при сбое системы. Зеленый используется для инсайдерских сборок, и я также встречал оранжевый.*

Систему можно настроить на выполнение некоторых операций в случае сбоя системы. Это можно сделать с помощью пользовательского интерфейса свойств системы на вкладке «Дополнительно». Щелкнув Настройки ...

Откроется диалоговое окно «Запуск и восстановление», в котором в разделе «Системный сбой» отображаются доступные параметры.

На рисунке ниже эти два диалоговых окна.



В случае сбоя системы запись о событии может быть записана в журнал событий. По умолчанию он установлен, и нет веских причин менять это.

Система настроена на автоматический перезапуск; это было по умолчанию с Windows 2000.

Самая важная настройка - создание файла дампа. Файл дампа фиксирует состояние системы во время сбоя, поэтому его можно позже проанализировать, загрузив файл дампа в отладчик.

Тип файла дампа важен, так как он определяет, какая информация будет присутствовать в дампе.

Важно подчеркнуть, что дамп не записывается в целевой файл во время сбоя, а вместо этого записывается в файл первой страницы файла подкачки. Только при перезагрузке системы ядро замечает, что есть дамп и информация в файле подкачки копирует данные в целевой файл. Причина связана, что во время сбоя системы может быть слишком опасно записывать что-либо в новый файл; система может быть недостаточно стабильна. Лучше всего записать данные в файл подкачки, который уже открыт.

Обратной стороной является то, что файл подкачки должен быть достаточно большим, чтобы содержать дамп, иначе файл дампа записываться не будет.

Вот варианты типа аварийного дампа:

- Небольшой дамп памяти - очень минимальный дамп, содержащий основную системную информацию и информация о ветке, вызвавшей сбой. Обычно этого слишком мало, чтобы определить, что случилось во всех случаях, кроме самых тривиальных. Плюс в том, что файл очень маленький.
- Дамп памяти ядра - это значение по умолчанию в Windows 7 и более ранних версиях. Эта настройка захватывает всю память ядра, но не память пользовательского режима. Обычно этого достаточно, поскольку сбой системы может быть вызван только некорректным поведением кода ядра. Крайне маловероятно, что пользовательский режим имел к этому какое-то отношение.
- Полный дамп памяти - обеспечивает дамп всей памяти, пользовательского режима и режима ядра.

Это наиболее полная доступная информация. Обратной стороной является размер дампа, который может быть гигантским в зависимости от ОЗУ системы и используемой в настоящее время памяти.

- Автоматический дамп памяти (Windows 8+) - это значение по умолчанию в Windows 8 и новее. Это то же, что и дамп памяти ядра, но ядро изменяет размер файла подкачки при загрузке до размера, который с высокой вероятностью гарантирует, что размер файла подкачки будет достаточно большим, чтобы содержать дамп ядра. Это возможно только в том случае, если размер файла подкачки указан как «Управляемый системой».
- Активный дамп памяти (Windows 10+) - аналогичен полному дампу памяти, за исключением того, что если в аварийной системе размещаются гостевые виртуальные машины, память, которую они использовали не фиксируется. Это помогает уменьшить размер файла дампа в серверных системах.

Информация о аварийном дампе

Получив аварийный дамп, вы можете открыть его в WinDbg, выбрав Файл / Открыть файл дампа.

Отладчик выдаст некоторую базовую информацию, подобную следующей:

```
***** Path validation summary *****
Response                               Time (ms)      Location
Deferred                               SRV*c:\Symbols*http://msdl.microsoft.\
com/download/symbols
Symbol search path is: SRV*c:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 10 Kernel Version 18362 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 18362.1.amd64fre.19h1_release.190318-1202
Machine Name:
Kernel base = 0xfffff803`70abc000 PsLoadedModuleList = 0xfffff803`70eff2d0
Debug session time: Wed Apr 24 15:36:55.613 2019 (UTC + 3:00)
System Uptime: 0 days 0:05:38.923
Loading Kernel Symbols
```

```
.....Page 2001b5efc too large to be in the dump file.
Page 20001ebfb too large to be in the dump file.
```

```
.....
Loading User Symbols
PEB is paged out (Peb.Ldr = 00000054`34256018). Type ".hh dbgerr001" for details
Loading unloaded module list
```

```
.....
For analysis of this file, run !analyze -v
nt!KeBugCheckEx:
fffff803`70c78810 48894c2408      mov     qword ptr [rsp+8],rcx ss:fffff988`53b0f6b0\
=000000000000000a
```

The debugger suggests running !analyze -v and it's the most common thing to do at the start of dump analysis. Notice the call stack is at KeBugCheckEx, which is the function generating the bugcheck, and is fully documented in the WDK, since drivers may also want to use it if necessary.

The default logic behind !analyze -v performs basic analysis on the thread that caused the crash and shows a few pieces of information related to the crash dump code:

```
2: kd> !analyze -v
*****
*
*                               Bugcheck Analysis                               *
*
*****
```

```
DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high. This is usually
caused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
Arguments:
Arg1: fffff907b0dc7660, memory referenced
Arg2: 0000000000000002, IRQL
Arg3: 0000000000000000, value 0 = read operation, 1 = write operation
Arg4: fffff80375261530, address which referenced memory
```

```
Debugging Details:
-----
```

```
(truncated)
```

```
DUMP_TYPE: 1
```

BUGCHECK\_P1: fffffd907b0dc7660

BUGCHECK\_P2: 2

BUGCHECK\_P3: 0

BUGCHECK\_P4: ffffff80375261530

READ\_ADDRESS: Unable to get offset of nt!\_MI\_VISIBLE\_STATE.SpecialPool  
Unable to get value of nt!\_MI\_VISIBLE\_STATE.SessionSpecialPool  
fffffd907b0dc7660 Paged pool

CURRENT\_IRQL: 2

FAULTING\_IP:

myfault+1530

fffff803`75261530 8b03 mov eax,dword ptr [rbx]

(truncated)

ANALYSIS\_VERSION: 10.0.18317.1001 amd64fre

TRAP\_FRAME: fffff98853b0f7f0 -- (.trap 0xffff98853b0f7f0)

NOTE: The trap frame does not contain all registers.

Some register values may be zeroed or incorrect.

rax=0000000000000000 rbx=0000000000000000 rcx=ffffd90797400340

rdx=0000000000000080 rsi=0000000000000000 rdi=0000000000000000

rip=fffff80375261530 rsp=fffff98853b0f980 rbp=0000000000000002

r8=ffffd9079c5cec10 r9=0000000000000000 r10=ffffd907974002c0

r11=ffffd907b0dc1650 r12=0000000000000000 r13=0000000000000000

r14=0000000000000000 r15=0000000000000000

iopl=0 nv up ei ng nz na po nc

myfault+0x1530:

fffff803`75261530 8b03 mov eax,dword ptr [rbx] ds:00000000`00000000=?\n  
???????

Resetting default scope

LAST\_CONTROL\_TRANSFER: from fffff80370c8a469 to fffff80370c78810

STACK\_TEXT:

fffff988`53b0f6a8 fffff803`70c8a469 : 00000000`0000000a fffffd907`b0dc7660 00000000`0\  
0000002 00000000`00000000 : nt!KeBugCheckEx

```

fffff988`53b0f6b0 fffff803`70c867a5 : ffff8788`e4604080 ffffff4c`c66c7010 00000000`0\
00000003 00000000`00000080 : nt!KiBugCheckDispatch+0x69
fffff988`53b0f7f0 fffff803`75261530 : ffffff4c`c66c7000 00000000`00000000 fffff988`5\
3b0f9e0 00000000`00000000 : nt!KiPageFault+0x465
fffff988`53b0f980 fffff803`75261e2d : fffff988`00000000 00000000`00000000 ffff8788`e\
c7cf520 00000000`00000000 : myfault+0x1530
fffff988`53b0f9b0 fffff803`75261f88 : ffffff4c`c66c7010 00000000`000000f0 00000000`0\
00000001 ffffff30`21ea80aa : myfault+0x1e2d
fffff988`53b0fb00 fffff803`70ae3da9 : ffff8788`e6d8e400 00000000`00000001 00000000`8\
3360018 00000000`00000001 : myfault+0x1f88
fffff988`53b0fb40 fffff803`710d1dd5 : fffff988`53b0fec0 ffff8788`e6d8e400 00000000`0\
00000001 ffff8788`ecdb6690 : nt!IofCallDriver+0x59
fffff988`53b0fb80 fffff803`710d172a : ffff8788`00000000 00000000`83360018 00000000`0\
00000000 fffff988`53b0fec0 : nt!IopSynchronousServiceTail+0x1a5
fffff988`53b0fc20 fffff803`710d1146 : 00000054`344feb28 00000000`00000000 00000000`0\
00000000 00000000`00000000 : nt!IopXxxControlFile+0x5ca
fffff988`53b0fd60 fffff803`70c89e95 : ffff8788`e4604080 fffff988`53b0fec0 00000054`3\
44feb28 fffff988`569fd630 : nt!NtDeviceIoControlFile+0x56
fffff988`53b0fdd0 00007ff8`ba39c147 : 00000000`00000000 00000000`00000000 00000000`0\
00000000 00000000`00000000 : nt!KiSystemServiceCopyEnd+0x25
00000054`344feb48 00000000`00000000 : 00000000`00000000 00000000`00000000 00000000`0\
00000000 00000000`00000000 : 0x00007ff8`ba39c147

```

(truncated)

FOLLOWUP\_IP:

myfault+1530

```
fffff803`75261530 8b03          mov     eax,dword ptr [rbx]
```

FAULT\_INSTR\_CODE: 8d48038b

SYMBOL\_STACK\_INDEX: 3

SYMBOL\_NAME: myfault+1530

FOLLOWUP\_NAME: MachineOwner

MODULE\_NAME: myfault

IMAGE\_NAME: myfault.sys

(truncated)

Каждый код аварийного дампа может иметь до 4 цифр, которые предоставляют дополнительную информацию о сбое.

В этом случае мы видим код DRIVER\_IRQL\_NOT\_LESS\_OR\_EQUAL (0xd1), а следующие четыре числа с именами от Arg1 до Arg4 означают (по порядку): указанная память, IRQL на момент вызов, операцию чтения и записи и адрес доступа.

Команда явно распознает myfault.sys как неисправный модуль (драйвер). Это потому, что это простой сбой - виновник находится в стеке вызовов, как можно увидеть в разделе STACK TEXT выше (вы также можно просто использовать команду k, чтобы увидеть это снова).

Более сложные аварийные дампы могут отображать вызовы от ядра только в стеке вызовов нарушившего поток. Прежде чем сделать вывод о том, что вы обнаружили ошибку в ядре Windows, примите во внимание следующее:

Скорее всего, драйвер сделал что-то, что само по себе не является фатальным, например, произошло переполнение буфера - записал данные за пределами выделенного буфера, но, к сожалению, следующая за этим буфером память была выделена под какой-то другой драйвер или ядро и ничего страшного тогда не произошло.

Некоторое время спустя ядро получило доступ к этой памяти, получило неверные данные и вызвало сбой системы. Но виновного драйвера нигде нет ни в одном стеке вызовов; это намного сложнее диагностировать.

### **Анализ файла дампа**

Файл дампа - это снимок системы. В остальном это тоже самое, что и любые сеансы отладки ядра.

Вы просто не можете устанавливать точки останова и, конечно же, не можете использовать какую-либо команду go. Все остальные команды доступны как обычно. Такие команды, как ! Process, ! Thread, !m, k, можно использовать как обычно.

Здесь будут приведены некоторые другие команды и советы:

- В подсказке указывается текущий процессор. Переключение процессоров может быть выполнено с помощью command ~ ns, где n - индекс ЦП (похоже на переключение потоков в пользовательском режиме).
- Команда! Running может использоваться для вывода списка потоков, которые выполнялись на всех процессорах в момент краха. Добавление -t в качестве опции показывает стек вызовов для каждого потока.

Вот пример с приведенным выше аварийным дампом:



2: kd> !running -t

System Processors: (000000000000000f)  
Idle Processors: (0000000000000002)

	Prcbs	Current	(pri) Next	(pri) Idle
0	fffff8036ef3f180	fffff8788e91cf080 ( 8)		fffff8037104840\
0	.....			

#	Child-SP	RetAddr	Call Site
00	00000094`ed6ee8a0	00000000`00000000	0x00007ff8`b74c4b57

2	fffffb000c1944180	fffff8788e4604080 (12)	fffffb000c195514\
0	.....		

#	Child-SP	RetAddr	Call Site
00	fffff988`53b0f6a8	fffff803`70c8a469	nt!KeBugCheckEx
01	fffff988`53b0f6b0	fffff803`70c867a5	nt!KiBugCheckDispatch+0x69
02	fffff988`53b0f7f0	fffff803`75261530	nt!KiPageFault+0x465
03	fffff988`53b0f980	fffff803`75261e2d	my fault+0x1530
04	fffff988`53b0f9b0	fffff803`75261f88	my fault+0x1e2d
05	fffff988`53b0fb00	fffff803`70ae3da9	my fault+0x1f88
06	fffff988`53b0fb40	fffff803`710d1dd5	nt!IofCallDriver+0x59
07	fffff988`53b0fb80	fffff803`710d172a	nt!IopSynchronousServiceTail+0x1a5
08	fffff988`53b0fc20	fffff803`710d1146	nt!IopXxxControlFile+0x5ca
09	fffff988`53b0fd60	fffff803`70c89e95	nt!NtDeviceIoControlFile+0x56
0a	fffff988`53b0fdd0	00007ff8`ba39c147	nt!KiSystemServiceCopyEnd+0x25
0b	00000054`344feb48	00000000`00000000	0x00007ff8`ba39c147

3	fffffb000c1c80180	fffff8788e917e0c0 ( 5)	fffffb000c1c9114\
0	.....		

#	Child-SP	RetAddr	Call Site
00	fffff988`5683ec38	fffff803`70ae3da9	Nt fs!Nt fsFsdClose
01	fffff988`5683ec40	fffff803`702bb5de	nt!IofCallDriver+0x59
02	fffff988`5683ec80	fffff803`702b9f16	FLTMGR!FltpLegacyProcessingAfterPreCallbacksC\ompleted+0x15e
03	fffff988`5683ed00	fffff803`70ae3da9	FLTMGR!FltpDispatch+0xb6
04	fffff988`5683ed60	fffff803`710cfe4d	nt!IofCallDriver+0x59
05	fffff988`5683eda0	fffff803`710de470	nt!IopDeleteFile+0x12d
06	fffff988`5683ee20	fffff803`70aea9d4	nt!ObpRemoveObjectRoutine+0x80
07	fffff988`5683ee80	fffff803`723391f5	nt!ObfDereferenceObject+0xa4
08	fffff988`5683eec0	fffff803`72218ca7	Nt fs!Nt fsDeleteInternalAttributeStream+0x111

```

09 fffff988`5683ef00 fffff803`722ff7cf Nt fs!Nt fsDecrementCleanupCounts+0x147
0a fffff988`5683ef40 fffff803`722fe87d Nt fs!Nt fsCommonCleanup+0xadf
0b fffff988`5683f390 fffff803`70ae3da9 Nt fs!Nt fsFsdCleanup+0x1ad
0c fffff988`5683f6e0 fffff803`702bb5de nt !Io ofCall Driver+0x59
0d fffff988`5683f720 fffff803`702b9f16 FLT MGR! FltpLegacyProcessingAfterPreCallbacksC\
ompleted+0x15e
0e fffff988`5683f7a0 fffff803`70ae3da9 FLT MGR! FltpDispatch+0xb6
0f fffff988`5683f800 fffff803`710ccc38 nt !Io ofCall Driver+0x59
10 fffff988`5683f840 fffff803`710d4bf8 nt !Io pCloseFile+0x188
11 fffff988`5683f8d0 fffff803`710d9f3e nt !ObCloseHandleTableEntry+0x278
12 fffff988`5683fa10 fffff803`70c89e95 nt !NtClose+0xde
13 fffff988`5683fa80 00007ff8`ba39c247 nt !KiSystemServiceCopyEnd+0x25
14 000000b5`aacf9df8 00000000`00000000 0x00007ff8`ba39c247

```

The command gives a pretty good idea of what was going at the time of the crash.

- The !stacks command lists all thread stacks for all threads by default. A more useful variant is a search string that lists only threads where a module or function containing this string appears. This allows locating driver's code throughout the system (because it may not have been running at the time of the crash, but it's on some thread's call stack). Here's an example for the above dump:

```

2: kd> !stacks
Proc.Thread .Thread Ticks ThreadState Blocker
[fffff803710459c0 Idle]
0.000000 fffff80371048400 00000003 RUNNING nt !KiIdleLoop+0x15e
0.000000 fffffb000c17b1140 0000ed9 RUNNING hal!HalProcessorIdle+0xf
0.000000 fffffb000c1955140 0000b6e RUNNING nt !KiIdleLoop+0x15e
0.000000 fffffb000c1c91140 000012b RUNNING nt !KiIdleLoop+0x15e
[fffff8788d6a81300 System]
4.000018 fffff8788d6b8a080 0005483 Blocked nt !PopFxEmergencyWorker+0x3e
4.00001c fffff8788d6bc5140 0000982 Blocked nt !ExpWorkQueueManagerThread+0x127
4.000020 fffff8788d6bc9140 000085a Blocked nt !KeRemovePriQueue+0x25c

```

(truncated)

```

2: kd> !stacks 0 myfault
Proc.Thread .Thread Ticks ThreadState Blocker
[fffff803710459c0 Idle]
[fffff8788d6a81300 System]

```

(truncated)

```

[fffff8788e99070c0 notmyfault64.e]
af4.00160c fffff8788e4604080 00000006 RUNNING nt !KeBugCheckEx

```

(truncated)

Адрес рядом с каждой строкой - это адрес ETHREAD потока, который может быть передан в поток!

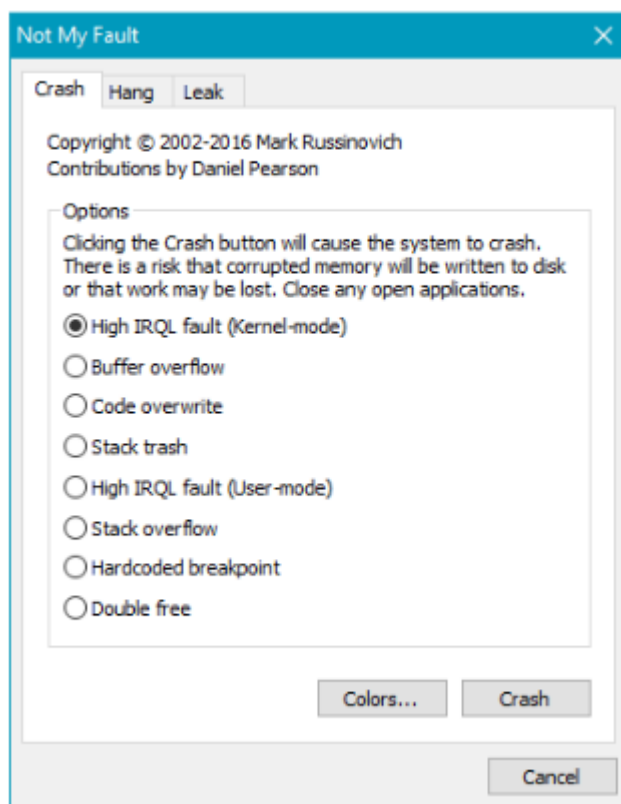
## Зависание системы

Сбой системы - это наиболее распространенный тип дампа, который обычно исследуется. Однако есть еще один тип дампа, с которым вам может понадобиться работать: зависшая система.

Как-же получить дамп такой системы ?

Если система все еще в какой-то степени реагирует, инструмент Sysinternals NotMyFault может вызвать сбой системы и, таким образом, принудительно создать файл дампа..

На рисунке ниже показан снимок экрана NotMyFault. Выбор первого (по умолчанию) варианта и щелчок по Crash немедленно вызывает сбой системы и создает файл дампа.



Если система полностью не отвечает, и вы можете подключить отладчик ядра, то можно затем выполнить отладку в обычном режиме или создайте файл дампа с помощью команды `.dump`.

Если система не отвечает, но не удастся подключить отладчик ядра, можно сгенерировать сбой вручную, если он настроен в реестре заранее, таким образом:

При обнаружении определенной комбинации клавиш драйвер клавиатуры, можно вызывать сбой.

Подробнее здесь: <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/forcing-a-system-crash-from-the-keyboard>

## **Синхронизация потоков**

Потокам иногда нужно согласовывать работу. Канонический пример - драйвер, использующий связанный список для сбора элементов данных.

Драйвер может быть вызван несколькими клиентами, поступающими из многих потоков в один или несколько процессов.

Это означает, что манипуляции со связанным списком должны выполняться атомарно. Если несколько потоков одновременно обращаются к одной и той же памяти, где хотя бы один является писателем (создание, изменение), это называется гонкой данных.

Обычно в драйвере из-за этого рано или поздно происходит сбой системы и повреждение данных практически гарантировано.

В таком сценарии важно, чтобы, пока один поток управлял связанным списком, все остальные потоки ожидали, пока первый поток закончит свою работу.

Это пример синхронизации потоков.

Ядро предоставляет несколько примитивов, которые помогают выполнить правильную синхронизацию для защиты данных во время одновременного доступа.

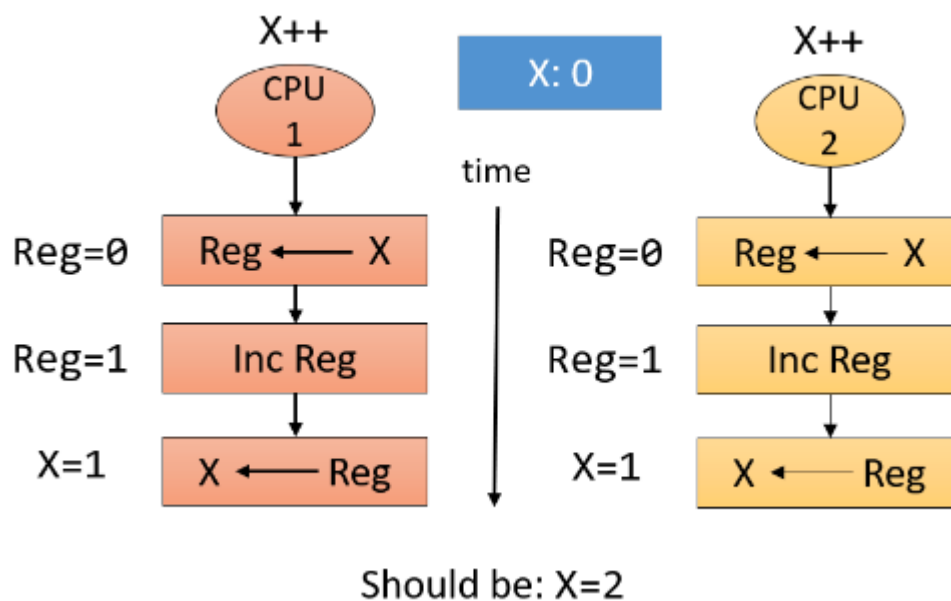
Ниже обсуждаются различные примитивы и методы для потоковой синхронизации.

## **Связанные операции**

Набор функций Interlocked обеспечивает удобные операции, которые выполняются атомарно, они используют оборудование, что означает отсутствие задействованных программных объектов.

Простой пример - увеличение целого числа на единицу. Как правило, это не атомарные операции.

Если два (или более) потока пытаются выполнить это одновременно в одном и том же месте памяти, это возможно (и вероятно), то некоторые приращения будут потеряны. На рисунке ниже показан простой сценарий, в котором увеличение значения на 1, выполненное из двух потоков, приводит к результату 1 вместо 2.



### Список основных Interlocke функций:

Function	Description
InterlockedIncrement / InterlockedIncrement16 / InterlockedIncrement64	Atomically increment a 32/16/64 bit integer by one
InterlockedDecrement / 16 / 64	Atomically decrement a 32/16/64 bit integer by one.
InterlockedAdd / InterlockedAdd64	Atomically add one 32/64 bit integer to a variable.
InterlockedExchange / 8 / 16 / 64	Atomically exchange two 32/8/16/64 bit values.
InterlockedCompareExchange / 64 / 128	Atomically compare a variable with a value. If equal exchange with the provided value and return TRUE; otherwise, place the current value in the variable and return FALSE.

### Объекты диспетчера

Ядро предоставляет набор примитивов, известных как объекты диспетчера, также называемые ожидаемыми объектами.

Эти объекты имеют состояние, называемое сигнальным и несигнальным, где значение сигнализируется и отсутствие сигнала зависит от типа объекта.

Они называются «ожидаемыми», потому что поток может ждать на таких объектах, пока они не станут сигнальными.

Во время ожидания поток не потребляет циклы процессора поскольку он находится в состоянии ожидания.

Основные функции, используемые для ожидания: KeWaitForSingleObject и KeWaitForMultipleObject.

Их прототипы (с упрощенными аннотациями SAL для ясности) показаны ниже:

**NTSTATUS KeWaitForSingleObject (**

```
_In_ PVOID Object,  
_In_ KWAIT_REASON WaitReason,  
_In_ KPROCESSOR_MODE WaitMode,  
_In_ BOOLEAN Alertable,  
_In_opt_ PLARGE_INTEGER Timeout);
```

**NTSTATUS KeWaitForMultipleObjects (**

```
_In_ ULONG Count,  
_In_reads_(Count) PVOID Object[],  
_In_ WAIT_TYPE WaitType,  
_In_ KWAIT_REASON WaitReason,  
_In_ KPROCESSOR_MODE WaitMode,  
_In_ BOOLEAN Alertable,  
_In_opt_ PLARGE_INTEGER Timeout,  
_Out_opt_ PKWAIT_BLOCK WaitBlockArray);
```

Вот краткое описание аргументов этих функций:

- **Object** - указывает ожидаемый объект. Обратите внимание, что эти функции работают с объектами, а не с дескрипторами. Если у вас есть дескриптор (может быть предоставлен пользовательским режимом), вызовите `ObReferenceObjectByHandle`, чтобы получить указатель на объект.
- **WaitReason** - указывает причину ожидания. Список причин ожидания довольно длинный, но драйверам следует обычно устанавливать для него значение "Executive", если он не ожидает из-за запроса пользователя, иначе укажите `UserRequest`.
- **WaitMode** - может быть `UserMode` или `KernelMode`. Большинство драйверов должны указывать `KernelMode`.
- **Alertable** - указывает, должен ли поток находиться в состоянии предупреждения во время ожидания. Состояние предупреждения позволяет доставить асинхронные вызовы процедур (APC) пользовательского режима. В пользовательском режиме APC могут доставляться, если режим ожидания - `UserMode`. Большинство драйверов должны указывать `FALSE`.
- **Timeout** - указывает время ожидания. Если указано `NULL`, ожидание неограниченно - до тех пор, пока требуется, чтобы объект стал сигнальным.

- Count - количество объектов ожидания.
- Object [] - массив указателей на объекты для ожидания.
- WaitType - указывает, следует ли ждать, пока все объекты будут сигнализированы одновременно (WaitAll) или всего один объект (WaitAny).
- WaitBlockArray - массив структур, используемых внутри для управления операцией ожидания. Это необязательно, если количество объектов  $\leq$  THREAD\_WAIT\_OBJECTS (в настоящее время 3) - ядро будет использовать встроенный массив, присутствующий в каждом потоке. Если количество объектов больше, драйвер должен выделить правильный размер структур из невыгружаемого пула и освободить их после ожидания.

Основные возвращаемые значения KeWaitForSingleObject:

- STATUS\_SUCCESS - ожидание удовлетворено, поскольку состояние объекта стало сигнальным.
- STATUS\_TIMEOUT - ожидание удовлетворено, поскольку истекло время ожидания.

Возвращаемые значения KeWaitForMultipleObjects поддерживают STATUS\_TIMEOUT так же, как KeWaitForSingleObject. STATUS\_SUCCESS возвращается, если указан тип ожидания WaitAll и все объекты стали сигнализировать. В случае ожидания WaitAny, если один из объектов стал сигнализированным, возвращаемое значение - это его индекс в массиве объектов.

В таблице ниже перечислены некоторые общие объекты диспетчера, а также сигнальные и несигнальные значения для этих объектов.

Object Type	Signaled meaning	Non-Signaled meaning
Process	process has terminated (for whatever reason)	process has not terminated
Thread	thread has terminated (for whatever reason)	thread has not terminated
Mutex	mutex is free (unowned)	mutex is held
Event	event is set	event is reset
Semaphore	semaphore count is greater than zero	semaphore count is zero
Timer	timer has expired	timer has not yet expired
File	asynchronous I/O operation completed	asynchronous I/O operation is in progress

В следующих разделах будут обсуждаться некоторые из общих типов объектов, полезных для синхронизации в драйверах.

Также будут обсуждаться некоторые другие объекты, которые не являются объектами диспетчера, но поддерживают ожидание для синхронизации потоков.

## Мьютекс

Мьютекс - классический объект для решения канонической проблемы доступа потоков к общему ресурсу в любое время.

Мьютекс сигнализируется, когда он свободен. Как только поток вызывает функцию ожидания и ожидание удовлетворяется, мьютекс становится несигнальным, и поток становится владельцем мьютекса.

Собственность очень важна для мьютекса.

Это означает следующее:

Если поток является владельцем мьютекса, то он единственный, кто может освободить мьютекс.

Использование мьютекса требует выделения структуры KMUTEX из невыгружаемого пула.

API мьютекса содержит следующие функции:

- KeInitializeMutex необходимо вызвать один раз для инициализации мьютекса.
- Одна из ожидающих функций, передающая адрес выделенной структуры KMUTEX.
- KeReleaseMutex вызывается, когда поток, являющийся владельцем мьютекса, хочет освободить его.

Учитывая вышеуказанные функции, вот пример использования мьютекса для доступа к некоторым общим данным, так что только один поток делает это за раз:

```
KMUTEX MyMutex;
```

```
LIST_ENTRY DataHead;
```

```
void Init() {
```

```
    KeInitializeMutex(&MyMutex, 0);
```

```
}
```

```
void DoWork() {
```

```
    // wait for the mutex to be available
```

```
    KeWaitForSingleObject(&MyMutex, Executive, KernelMode, FALSE, nullptr);
```

```
    // access DataHead freely
```

```
    // once done, release the mutex
```

```
    KeReleaseMutex(&MyMutex, FALSE);
```

```
}
```



Важно освободить мьютекс несмотря ни на что, поэтому лучше использовать `__try / __finally`, чтобы быть уверен, что это делается при любых обстоятельствах:

```
void DoWork() {  
    // wait for the mutex to be available  
    KeWaitForSingleObject(&MyMutex, Executive, KernelMode, FALSE, nullptr);  
    __try {  
        // access DataHead freely  
    }  
    __finally {  
        // once done, release the mutex  
        KeReleaseMutex(&MyMutex, FALSE);  
    }  
}
```

Поскольку использование `__try / __finally` немного неудобно, мы можем использовать C++ для создания оболочки RAII.

Это также может быть использовано для других примитивов синхронизации.

Сначала мы создадим оболочку мьютекса, которая предоставляет функции с именами `Lock` и `Unlock`:

```
struct Mutex {  
    void Init() {  
        KeInitializeMutex(&_mutex, 0);  
    }  
    void Lock() {  
        KeWaitForSingleObject(&_mutex, Executive, KernelMode, FALSE,  
        nullptr);  
    }  
    void Unlock() {  
        KeReleaseMutex(&_mutex, FALSE);  
    }  
private:  
    KMUTEX _mutex;  
};
```

```

template<typename TLock>
struct AutoLock {
    AutoLock(TLock& lock) : _lock(lock) {
        lock.Lock();
    }
    ~AutoLock() {
        _lock.Unlock();
    }
private:
    TLock& _lock;
};

```

Затем мы можем создать общую оболочку RAII для ожидания любого типа, который имеет блокировку и разблокировку:

```

template<typename TLock>
struct AutoLock {
    AutoLock(TLock& lock) : _lock(lock) {
        _lock.Lock();
    }
    ~AutoLock() {
        _lock.Unlock();
    }
private:
    TLock& _lock;
};

```

Имея эти определения, мы можем заменить код, использующий мьютекс, следующим:

```
void Init() {  
    MyMutex.Init();  
}  
  
void DoWork() {  
    AutoLock<Mutex> locker(MyMutex);  
    // access DataHead freely  
}
```

Мы будем использовать тот же тип AutoLock и с другими примитивами синхронизации.

### Fast Mutex

Быстрый мьютекс - это альтернатива классическому мьютексу, обеспечивающая лучшую производительность. Это не диспетчер объекта и, следовательно, имеет собственный API для получения и освобождения мьютекса.

Он имеет следующие характеристики по сравнению с обычным мьютексом:

- Быстрый мьютекс не может быть получен рекурсивно.
- При получении быстрого мьютекса IRQL ЦП повышается до APC\_LEVEL (1).
- Быстрый мьютекс можно ожидать только бесконечно - нет способа указать тайм-аут.

Из-за первых двух пунктов выше, быстрый мьютекс немного быстрее, чем обычный мьютекс. По факту, большинство драйверов, требующих мьютекса, используют быстрый мьютекс, если нет веских причин использовать обычный мьютекс.

Быстрый мьютекс инициализируется путем выделения структуры FAST\_MUTEX из невыгружаемого пула и вызовом ExInitializeFastMutex.

Получение мьютекса выполняется с помощью ExAcquireFastMutex или ExAcquireFastMutexUnsafe (если текущий IRQL уже равен APC\_LEVEL).

Освобождение быстрого мьютекса выполняется с помощью ExReleaseFastMutex или ExReleaseFastMutexUnsafe.

**Быстрый мьютекс не доступен в пользовательском режиме. Код пользовательского режима может использовать только обычный мьютекс.**

С точки зрения общего использования быстрый мьютекс эквивалентен обычному мьютексу. Просто немного быстрее.

Мы можем создать оболочку C++ над быстрым мьютексом, так что его получение и освобождение может автоматически достигаться с помощью класса AutoLock RAII, определенного в предыдущем разделе:

```
// fastmutex.h

class FastMutex {
public:
    void Init();
    void Lock();
    void Unlock();
private:
    FAST_MUTEX _mutex;
};

// fastmutex.cpp

#include "FastMutex.h"

void FastMutex::Init() {
    ExInitializeFastMutex(&_mutex);
}

void FastMutex::Lock() {
    ExAcquireFastMutex(&_mutex);
}

void FastMutex::Unlock() {
    ExReleaseFastMutex(&_mutex);
}
```

## Семафор

Основная цель семафора - ограничить что-то, например длину очереди. Семафор инициализируется максимальным и начальным счетчиком (обычно устанавливается на максимальное значение) путем вызова KeInitializeSemaphore. Пока его внутренний счетчик больше нуля, семафор сигнализируется.

Поток, который вызывает KeWaitForSingleObject, удовлетворяет ожидание и счетчик семафоров уменьшается на единицу.

Это продолжается до тех пор, пока счетчик не достигнет нуля, после чего семафор перестанет работать.

В качестве примера представьте себе очередь рабочих элементов, управляемую драйвером. Некоторые объекты хотят добавить элементы в очередь. Каждый такой поток вызывает KeWaitForSingleObject для получения одного «счетчика» семафора. Пока счетчик больше нуля, поток продолжает и добавляет элемент в очередь, увеличивая ее длину, и семафор «теряет» счет. Некоторым другим потокам поручено обработка заданий из очереди. Как только поток удаляет элемент из очереди, он вызывает KeReleaseSemaphore, который увеличивает счетчик семафора, переводя его в сигнальное состояние снова, позволяя потенциально другому потоку добиться прогресса и добавить новый элемент в очередь.

## **События**

Событие инкапсулирует логический флаг - истинный (сигнальный) или ложный (несигнальный).

Главная цель события - сигнализировать о том, что что-то произошло, обеспечить синхронизацию потока.

Например, если какое-то условие становится истинным, может быть установлено событие и может быть выпущена группа потоков от ожидания и продолжить работу с некоторыми данными, которые, возможно, теперь готовы к обработке.

Есть два типа событий, тип указывается во время инициализации события:

- Событие уведомления (ручной сброс) - когда это событие установлено, оно освобождает любое количество потоков, и состояние события остается установленным (сигнализируемым) до явного сброса.
- Событие синхронизации (автоматический сброс) - когда это событие установлено, оно освобождает не более одного потока и после освобождения событие возвращается к сбросу (Несигнальное состояние) автоматически.

Событие создается путем выделения структуры KEVENT из невыгружаемого пула и последующего вызова KeInitializeEvent для его инициализации, указав тип события (NotificationEvent или SynchronizationEvent) и начальное состояние события (сигнальное или несигнальное).

Ожидание события окончено обычно с функциями KeWaitXxx. Вызов KeSetEvent устанавливает событие в сигнальное состояние, при вызове KeResetEvent или KeClearEvent его сбрасывает (несигнальное состояние) (последняя функция немного быстрее, так как не возвращает предыдущее состояние события).

## Ресурсы исполнения

С помощью мьютекса или быстрого мьютекса решена классическая проблема синхронизации доступа к общему ресурсу несколькими потоками.

Это работает, но мьютексы позволяют давать лишь один доступ к ресурсу для потока. Это может быть неудачно в тех случаях, когда несколько потоков хотят получить доступ общему ресурсу только для чтения.

В случаях, когда можно отличить изменение данных (запись) от простого просмотра данных (чтения) - есть возможная оптимизация.

Поток, которому требуется доступ к общему ресурсу, может объявить свои намерения - читать или писать.

Если он объявляет чтение, другие потоки, объявляющие чтение, могут делать это одновременно, что повысит производительность. Это особенно полезно, если общие данные меняются медленно, т. е. значительно больше читают, чем пишут.

Ядро предоставляет еще один примитив синхронизации, ориентированный на этот сценарий, известен как один писатель, несколько читателей.

Этот объект — Ресурс исполнения, еще один специальный объект, который не является объектом диспетчера.

Инициализация ресурса исполнения выполняется путем выделения структуры `ERESOURCE` из невыгружаемого пула и вызов `ExInitializeResourceLite`.

После инициализации потоки могут получить специальную блокировку (для записи) с использованием `ExAcquireResourceExclusiveLite` или общую блокировку путем вызова `ExAcquireResourceSharedLite`.

После выполнения работы поток освобождает исполнительный ресурс с `ExReleaseResourceLite`.

Следующий фрагмент кода демонстрирует это:

```
ERESOURCE resource;
```

```
void WriteData() {  
    KeEnterCriticalRegion();  
  
    ExAcquireResourceExclusiveLite(&resource, TRUE);  
  
    // wait until acquired  
  
    // Write to the data  
  
    ExReleaseResourceLite(&resource);  
  
    KeLeaveCriticalRegion();  
}
```

Поскольку эти вызовы настолько распространены при работе с исполнительными ресурсами, есть функции, которые выполнить обе операции за один вызов:

```
void WriteData() {  
    ExEnterCriticalRegionAndAcquireResourceExclusive(&resource);  
    // Write to the data  
    ExReleaseResourceAndLeaveCriticalRegion(&resource);  
}
```

## Синхронизация с высоким IRQL

До сих пор разделы о синхронизации касались потоков, ожидающих различных типов объектов.

Однако в некоторых сценариях потоки не могут ждать - в частности, когда IRQL процессора DISPATCH\_LEVEL (2) или выше. В этом разделе обсуждаются эти сценарии и способы их устранения.

Давайте рассмотрим пример сценария: драйвер имеет таймер, настроенный с помощью KeSetTimer, и использует DPC для выполнения кода по истечении таймера. В то же время другие функции в драйвере, такие как IRP\_MJ\_DEVICE\_CONTROL может выполняться одновременно (выполняется с IRQL 0).

Если обе эти функции необходимо получить доступ к общему ресурсу (например, связанному списку), они должны синхронизировать доступ, чтобы предотвратить потери данных.

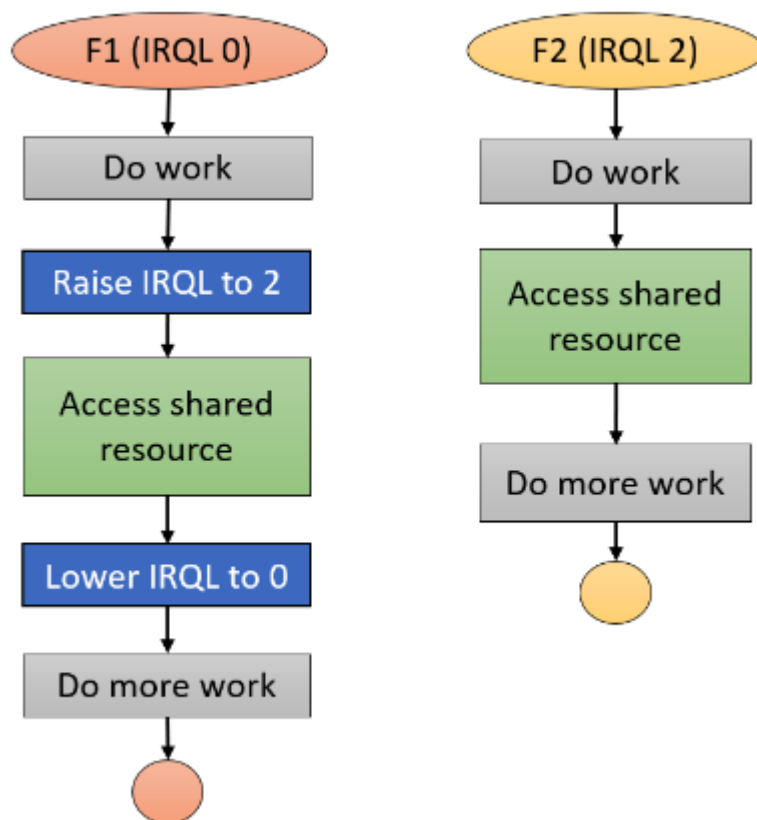
Проблема в том, что DPC не может вызвать KeWaitForSingleObject или любую другую функцию ожидания. Итак, как эти функции могут синхронизировать доступ?

В простом случае система имеет один процессор. В этом случае при доступе к общему ресурсу, функция низкого IRQL просто должна поднять IRQL до DISPATCH\_LEVEL, а затем получить доступ к ресурсу.

В это время DPC не может вмешиваться в этот код, поскольку IRQL ЦП уже равен 2. Один раз код выполняется с общим ресурсом, он может снизить IRQL до нуля, позволяя DPC выполнить.

Это предотвращает одновременное выполнение этих подпрограмм.

На рисунке ниже показан этот принцип:



В стандартных системах, где имеется более одного процессора, этого метода синхронизации недостаточно, потому что IRQL - это свойство ЦП, а не общесистемное свойство.

Если IRQL одного процессора повышается до 2, если DPC необходимо выполнить, он может помешать другому ЦП, чей IRQL может быть равен нулю.

Как мы можем это решить?

Такой объект действительно существует - Spin Lock.

### Spin Lock

Spin Lock - это простой бит в памяти, который обеспечивает атомарные операции проверки и изменения через API.

Перед работой с разделяемым ресурсом процессор записывает бит в памяти, что означает, что ресурс занят и используется процессором, другой процессор просто будет ожидать снятия данного бита.

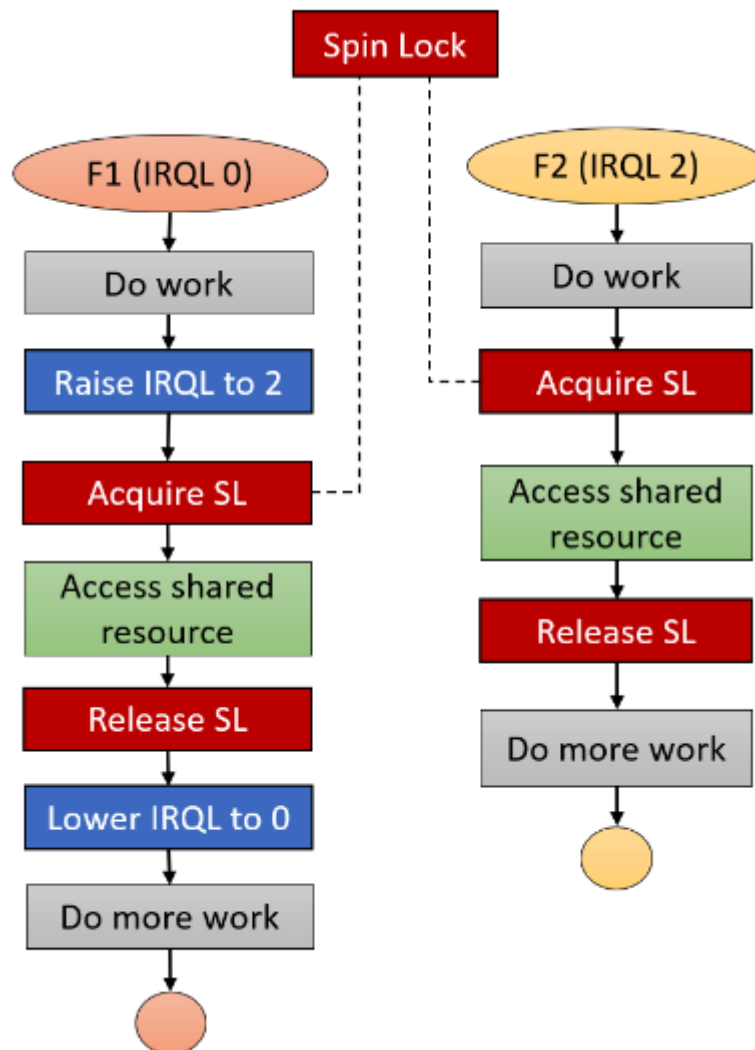
В сценарии, описанном выше, потребуется выделить и инициализировать спин-блокировку.

Каждая функция, которая требует доступа к общим данным, должна поднять IRQL до 2 (если еще не было), получить блокировку, выполнить работу с общими данными и,



наконец, снять блокировку и понизить IRQL обратно (если применимо не для DPC). Эта цепочка событий показана на рисунке ниже.

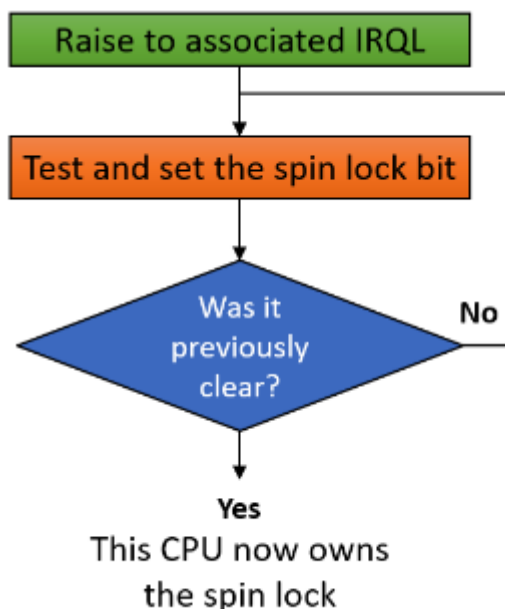
Создание спин-блокировки требует выделения структуры KSPIN\_LOCK из невыгружаемого пула и вызова KeInitializeSpinLock. Это переводит спин-блокировку в состояние без владельца.



Получение спин-блокировки - это всегда двухэтапный процесс: во-первых, поднимите IRQL до нужного уровня, т. е. самый высокий уровень любой функции, пытающейся синхронизировать доступ к общему ресурсу.

Во-вторых, получите спин-блокировку. Эти два шага объединены используя соответствующий API.

Этот процесс изображен на следующем рисунке.



## Рабочие элементы

Иногда возникает необходимость запустить часть кода в потоке, отличном от выполняемого.

Один способ сделать это - явно создать поток и поручить ему выполнение кода.

Ядро предоставляет функции, которые позволяют драйверу создавать отдельный поток выполнения: `PsCreateSystemThread` и `IoCreateSystemThread` (доступно в Windows 8+).

Эти функции подходят, если драйвер нужно долгое время запускать код в фоновом режиме.

Однако для операций с ограниченным сроком лучше использовать предоставленный ядром пул потоков, который будет выполнять ваш код в некотором системном рабочем потоке.

Рабочие элементы - это термин, используемый для описания функций, поставленных в очередь в системный пул потоков.

Драйвер может выделить и инициализировать рабочий элемент, указав на функцию, которую драйвер хочет выполнить, а затем рабочий элемент можно поставить в очередь в пул.

Это очень похоже на DPC, основное отличие поскольку рабочие элементы всегда выполняются на IRQL PASSIVE\_LEVEL, то есть этот механизм можно использовать для выполнения операций с IRQL 0 из функций, работающих с IRQL 2.

Например, если подпрограмма DPC необходимо выполнить операцию, которая не разрешена на IRQL 2 (например, открытие файла), он может использовать рабочий элемент для выполнения этих операций.

Создать и инициализировать рабочий элемент можно одним из двух способов:

- Выделите и инициализируйте рабочий элемент с помощью IoAllocateWorkItem. Функция возвращает указатель на IO\_WORKITEM. По завершении рабочего элемента его необходимо освободить с помощью IoFreeWorkItem.
- Динамически распределять структуру IO\_WORKITEM с размером, предоставленным IoSizeofWorkItem.

Затем вызовите IoInitializeWorkItem. Когда закончите с рабочим элементом, вызовите IoUninitializeWorkItem.

Эти функции принимают объект устройства, поэтому убедитесь, что драйвер не выгружен, пока идет работа.

Чтобы поставить рабочий элемент в очередь, вызовите IoQueueWorkItem. Вот его определение:

```
viud IoQueueWorkItem(  
_____Inout_ PIO_WORKITEM IoWorkItem,  
_____In_ PIO_WORKITEM_ROUTINE WorkerRoutine,  
_____In_ WORK_QUEUE_TYPE QueueType,  
_____In_opt_ PVOID Context);
```

Функция обратного вызова, которую должен предоставить драйвер, имеет следующий прототип:

```
IO_WORKITEM_ROUTINE WorkItem;  
void WorkItem(  
_____In_  
_____PDEVICE_OBJECT DeviceObject,  
_____In_opt_ PVOID  
_____Context);
```

В системном пуле потоков есть несколько очередей, в зависимости от приоритетов потоков, которые обслуживают эту работу.

Здесь показано несколько уровней:

```
typedef enum _WORK_QUEUE_TYPE {  
    CriticalWorkQueue,  
    // priority 13  
    DelayedWorkQueue,  
    // priority 12  
    HyperCriticalWorkQueue,  
    // priority 15  
    NormalWorkQueue,  
    // priority 8  
    BackgroundWorkQueue,  
    // priority 7  
    RealTimeWorkQueue,  
    // priority 18  
    SuperCriticalWorkQueue,  
    // priority 14  
    MaximumWorkQueue,  
    CustomPriorityWorkQueue = 32  
} WORK_QUEUE_TYPE;
```

В документации указано, что следует использовать DelayedWorkQueue, но на самом деле любой другой поддерживаемый уровень можно использовать.

## Резюме

В этой главе мы рассмотрели различные механизмы ядра, о которых следует знать разработчикам драйверов.

В следующей главе мы более подробно рассмотрим пакеты запросов ввода-вывода (IRP).