Windows NT File System Internals

第一部分

第三章　结构化的驱动开发

写一个内核驱动不容易。不幸的是，在你的生产系统上安装一个新的内核驱动有时候是很糟糕的事情。驱动作为 NT 执行体的一部分能够使你的系统崩溃，而且定位负责的模块非常困难，另外，系统崩溃可能很有规律，也可能完全偶然（通常，当你在做很重要的事情的时候只有虔诚地祈祷了）。更坏的是，内核驱动可能破坏你的数据，而且在你发现数据已经被破坏的时候再来恢复数据已经太晚了。

因此，如果你正在安装一个新内核驱动，你对这个驱动应该有些期望，比如：

这个驱动不会造成数据丢失，这是内核驱动设计和开发人员最基本的责任，不幸的是，客观地评价的话这也是最难做到的。

驱动不应该引起系统崩溃，目标是保证在不利的情况下，当外部连接的设备（比如磁盘驱动或者网卡）没有正常工作的时候，驱动程序必须文雅地处理这些错误。注意，文雅地处理一个错误是有些模糊的：他可能表示驱动应该在尽可能的各种情况中工作。也可能意味着驱动应该把自己停掉（比如不提供相关的功能），但是允许操作系统的其他部分继续正常工作。

在前一点上更深地说，驱动代码中的软件错误（不可避免的即使开发者考虑了异常情况）不应该导致系统崩溃。这看起来很荒谬，因为 BUGS 的定义是不可预期的，所以很难预料和管理。但是驱动开发这很多情况下是有可能去准备对付不可预料的在代码中的错误和在自我测试中没有发现的软件错误的。如果最后的驱动被正确地实现，确实可能做到的，那么在大多数情况下，那样的 BUGS 就不会导致系统崩溃。

驱动应该能够提供适当的状态信息给系统管理员，例如，错误发生的情况下，严格清晰，简要的问题说明应该被驱动转达到管理员，允许管理员试图矫正问题。或者知道某些功能缺失了或者可能要发生了。甚至当驱动和设备正常工作的时候，也可以有适当的状态报告给管理员。当错误发生时候驱动提供的数据可能包含恢复错误时的信息，某些性能相关的统计数据或者自动调整的驱动的参数的值。这些可以帮制管理员理解系统的行为和局限然后可能也给他们机会去修改工作负载或者在基于期望的用法模式来调整驱动的参数。

完成用户的这些期望的责任直接落到了内核驱动设计和开发人员的身上。无论如何，好消息是为 NT 平台开发达到这些愿望的软件是可能的；实际上，操作系统为开发人员提供了丰富的支持来在他们的驱动中达到这些特征。

异常分派支持

异常是在一个线程执行一些指令的时候发生的非典型的事件。异常在导致异常的线程的上下文中进行同步处理。因为异常情况是在指令执行时直接产生的同步事件，他们可以再生，提供确实一样的情况可以被重复和重复的指令。NT 内核提供当遇到异常时候的异常分派支持。

1. 为线程创建一个陷阱帧，其中包括所有可变寄存器的内容等，寄存器的内容在处理异常的时候可能被覆盖。
2. 可以创建一个异常帧，其中包含其他不便的寄存器的内容。陷阱处理器模块总是在处理异常的时候创建异常帧。
3. 创建一个异常记录结构，其中包含描述发生的异常的异常代码，异常标志，发生异常的代码的地址，以及其他的和特定异常情况相关的参数。当前有效的异常标志的值是

EXCEPTION_NONCONTINUABLE，指出这是一个致命异常，进一步的处理应该结束。一些异常还有由陷阱处理器提供的关于异常情况的更多的信息的附加参数。唯一的有附加参数的异常情况是EXCEPTION_ACCESS_VIOLATION，提供两个相关参数，一个指出导致访问违例的操作是读还是写，另一个是不能访问到的虚拟地址。

4. 控制交给NT内核的异常分派器模块。内核中的异常分派模块叫KiDispatchException()。

处理异常的可能的结果

异常分派器要决定异常发生时候的处理器模式。用户模式的异常和内核模式的异常在处理上稍稍有些差别，但是基本的原理是一样的。在讲述异常处理器在处理中采取的步骤之前，我们简单讨论异常处理可能的结果

每个异常可能使用三种方式中的一种进行处理：

异常处理器改变导致异常的一个或者多个条件然后让异常分派器从新试图执行指令。

例如，假设一个页错误的异常发生了。异常处理器将会调用页错误处理器把需要的页从辅助存储器或者网络换进系统内存中。现在重新试图访问内存应该就会成功了。

另一个例子是当一段代码试图分配有些内存接着来使用。如果分配内存失败了，访问指向那个内存块的指针就会引起内存访问违例。下面的指令示意这种情况：

```
// allocate 4K bytes
SomePtr = ExAllocatePool(PagedPool, 4096);
// Normally, the memory allocation request will succeed and SomePtr
// will contain a valid pointer address. However, it is possible that
// the request may occasionally fail. For the sake of discussion,
// assume that in the particular instance described below, the
// memory allocation request does fail and therefore ExAllocatePool()
// returns NULL.
// Although I personally recommend always checking the value of the
// returned pointer value above, many other designers might argue
// that structured exception handling allows for more readable
// code by avoiding unnecessary, multiple, if (...) {}, kind of
// statements.
RtlZeroMemory(SomePtr, 4096);
// If SomePtr was NULL, we'll get an exception in the statement above.
```

用NULL 指针来调用RtlZeroMemory ()将导致一个EXCEPTION_ACCESS_VTOLATION异常的产生，异常处理器可以可以把指针指向某些以前分配的内存再重新执行指令。

警告：重新执行RtlZeroMemory ()（在这里是）相应的汇编代码可能导致未预期的错误。除非已编译的汇编代码经过仔细检查，你不能保证在异常处理器中改变SomePtr的值有期望的结果。举例来说，编译器可能用SomePtr的值来初始化了一个寄存器，这里是NULL。现在装有值的寄存器将被重复使用，因为在哪个寄存器后面的指令即会重新执行的指令可能是内存移动（*move memory*）指令。这意味着在异常处理器中重新对SomePtr赋值没有作用，异常会永远不断产生。因此重新执行导致 异常的指令最好谨慎处理。

如果希望异常处理器可以返回一个指示异常发生时候重新执行的指令的执行结果。记住常量值实际是不重要的，但是返回这样的一个值的事实是值得注意的。

异常处理器判断出这个异常不是他要处理的。

如果这种情况发生了，那么一个适当的值将会返回指出这个异常应该被继续传递。因此，异常分派器会继续寻找其他可能希望处理这个异常的异常处理器。

例如，假设一个特定的只处理EXCEPTION_ACCESS_VIOLATION类型的异常的异常处理器。

如果一个数据对齐的异常发生了，那么这个异常处理器会返回一个值指出他不能处理这个异常，分派器应该继续遍历调用帧来寻找一个准备处理这种异常的异常处理器。

异常处理器执行一块特殊的代码来处理异常，然后向调用者指出应该在异常处理器后面接着执行代码。

在这种情况下，异常处理器不会重试导致异常的代码，而是试图在异常处理器执行后立刻继续执行指令。

这种处理异常的方法和前面描述的简单地修改一些条件然后重新执行指令是有很大的区别的。这里，异常处理器执行一些处理然后继续在异常处理器代码后面立即执行指令。这种情况下导致异常的条件已经被修改了（前面描述的）。异常处理器希望在上一次引起异常的指令的地方开始执行。

异常处理器不必存在于引起异常的系统的函数或者过程中。他可以包含在调用层次的任何例程里。

例如，假设一种情况：*procedure_A* 调用 *procedure_B, procedure_B又调用 procedure_C。更进一步，假设一个*异常（EXCEPTION_ACCESS_VIOLATION）在*procedure_C*中发生了。很可能在*procedure_B和 procedure_C*中都没有处理这种异常的处理器。

如果异常处理器在*procedure_A*中，如果*procedure_A中的处理器处理了这个异常，执行流将从procedure_A中*异常处理器后面的指令继续。*procedure_B和 procedure_C的栈将会自动弹出以便在procedure_A中继续*。图3-1示意这种情况。

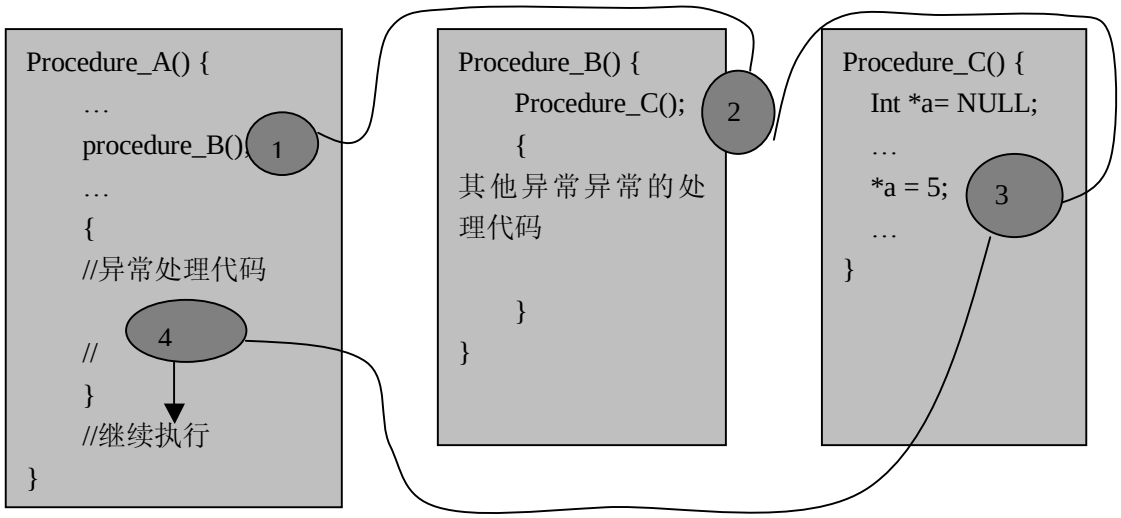

图 3-1 异常处理器处理一个异常后的执行流程

正如你看见的，异常处理器可以有三种不同的方式来响应一个特定的异常情况。

晚一点会讨论的，procedure_B和Procedure_C弹出的栈会引起适当的终结处理器被调用。这就允许在那些过程中的系统出栈，从而防止可能发生的讨厌的比如死锁这样的副作用在控制流从procedure_C到procedure_A的转换中的不想看见的结果。

分派内核异常

给定异常处理器在处理异常时候的不同的途径，有助于理解内核中的分派器代码KiDispatchException()究竟怎样处理异常情况。下面是当一个内核线程代码发生异常而调用KiDispatchException()的时候它会做的步骤：

1．首先，异常分派器检查调试器是不是激活的，如果是的话就把控制交给调试器。

调试器会用返回值TRUE指出异常已经处理了，或者异常没有处理，应该继续寻找其他的异常处理器。

注意调试器可能从传给他的执行上下文结构中得到当前指令指针或者当前栈指针并修改他们，因

此，如果异常已经被调试器处理了，就没有必要从导致异常的指令重新继续执行了。
如果调试器返回TRUE指出异常已经处理了，KiDispatchException()通过调用分派器的陷阱处理器把控制返回给发生异常的线程。
2．如果调试器没有出现或者调试器返回FALSE指出异常没有处理，分派器会试图调用基于调用帧的任何异常处理器。
调用基于调用帧的异常处理器是通过一个叫RtlDispatchException()的RTL函数，这个函数通常不暴露给第三方开发人员。
RtlDispatchException()
函数向相反方向搜索基于栈的调用帧来寻找一个准备处理这个异常的异常处理器。搜索直到某个异常处理器返回值指出应该重试导致异常的代码或者检查了整个调用层次但没有发现适当的异常处理器。
NT平台的编译器支持异常处理机制使用这个RTL为执行线程注册异常处理器。不幸的是，注册和取消注册异常处理器的函数一般不向第三方开发人员暴露。但是好消息是内核驱动极少直接访问这些函数，因为编译器通常提供结构化异常处理机制为你执行那些烦琐的工作。
RTL包还为编译器开发人员提供为执行线程注册终止处理器的接口。终止处理器在这章的后面描述。
3． 在第2步中，基于调用帧的异常处理器可能找到，可能找不到。
即使一个或者多个异常处理器被找到，但是那些处理器可能不准备处理这种异常情况。在这节后面描述的，结构化异常允许你检查异常的类型然后决定你是否希望在你的异常处理器中处理这个异常，或者你希望把这个异常传给下一个可能的异常处理器。
如果RtlDispatchException()返回FALSE指出异常没有处理，异常处理器再次试图调用任何可以执行的调试器。这叫做第二次机会处理，在这一点这个动作保证将会产生一个和发生了的异常的一样的回归的异常发生。
如果调试器是连通的，就有一个最后的机会救活系统。但是，如果没有连通的调试器或者调试器再一次返回FALSE，异常处理器将调用KeBugCheck ()，停止整个系统，结果就是可怕的蓝屏死机。
系统崩溃的原因将是KMODE_EXCEPTION_NOT_HANDLED，指出在内核执行的时候发生的异常找不到异常处理器来处理。除非极端的考虑，你的代码决不应该引起蓝屏。
异常分派支持，还有基于调用帧的异常处理注册是NT操作系统提供的而不是编译器提供的功能。也就是说，除非你使用支持NT异常处理模式的编译器，不然你不能使用操作系统支持的异常处理的特性。MICROSOFT C/C++编译器支持用户模式和内核模式的结构化异常处理。
异常分派器：用户模式异常
由KiDispatchException( ) 直接调用RtlDispatchException()来处理异常是在内核中执行的代码发生异常时候的的处理方法。如果异常发生在用户模式，KiDispatchException( )执行的处理稍稍有些不一样。
使用LPC把一个消息送到进程的调试端口。如果这个端口处理了这个异常，KiDispatchException( )就没有其他的事情可做了。
可是，考虑一下这种情况：进程的调试端口处理异常失败。试图在内核中执行用户模式的异常处理器不是一件明智的事情。最低限度，这会给操作系统引入一个大的安全漏洞。因此，
KiDispatchException( )函数把控制交给适当的用户空间的异常分派模块。这个分派模块把陷阱帧，异常帧，和异常记录压栈到用户空间的栈上。然后KiDispatchException( )修改异常记录等，一旦控制从异常分派器和陷阱处理器返回， 特定的用户空间的过程会被调用来处理异常的更多的工作。会调用任何用户定义的异常处理器。注意这里的修改处理包括修改异常记录中的指令指针来指向用户空间的异常分派器函数。

处理用户模式的异常也是相似的：检查调用层来寻找一个能处理这个异常的异常处理器。如果没有用户模式的异常处理器处理这个异常，包含引起异常的线程通常被结束掉。结束用户模式的线程是由默认异常处理器来做的，通常由WIN32或其他子系统安装。
现在明白了当异常发生时候的处理过程，下一个合理的问题是：该怎样写代码才能处理未预期的事件，比如异常？下一节就给出一个答案。

结构化异常处理（SHE）
WINDOWS NT执行体中大量使用SEH，每个NT内核组件都试着处理执行中发生的预料之外的事件，这些模块执行的时候很难保证任何异常事件不会导致整个系统停掉。独立驱动的开发人员在他们的驱动中实现SHE是一个极好的习惯。这会产生更健壮的内核驱动，更稳定的WINDOWS NT系统，给用户更多快乐。

> 提示：作为一个驱动开发人员，你可以选择在你的驱动里避免使用SHE，许多驱动程序就是这样做的而且作的很好。但是，如果你开发文件系统驱动，我强烈要求你使用SEH。不仅因为这样做是正确的做法，而且有些NT CATCH MANAGER支持例程和有些VMM函数在有情况下会触发异常来代替返回错误，以便不造成系统的紊乱（恐慌？PANIC）。文件系统驱动将会处理这些异常，并把他们作为正常的错误情况。
>
> 如果你的驱动中使用SHE，你可以温和地处理这些异常，不使用异常处理将导致本可避免的BUGCHECK状态。

在讨论什么是SHE和他能提供什么之前，我先告诉你SHE不能提供的。
SHE不是糟糕的驱动设计或者实现的万能灵药。如果你没有仔细设计和实现驱动，SHE也不能改变这种情况。同样的，SHE也不能保证能完全避免系统崩溃，在DISPATCH_LEVEL或者更高IRQL访问分页内存必然导致系统崩溃，而不管异常处理有没有。
最后，SHE应该普遍的存在于系统实现中以便得到坚实的好处。如果异常处理没有系统地贯穿驱动程序的代码，那么实现在那些没有受到SHE保护的部分依然容易受到未知错误情况的威胁。
SHE是一个方法，开发和设计人员可以提供异常处理那些异常情况，避免KiDispatchException ( )调用默认处理，即是调用KeBugCheck ( )。
注意：没有处理所有内核异常的默认处理器存在。因此，如果你的文件系统或者过滤驱动导致一个异常但是没有任何异常处理器能处理哪个异常的话，就会导致系统崩溃。
而且，SHE允许设计者提供从特定的代码快中跳出的系统的方法。这个跳出能够帮助保证异常情况不会导致死锁，挂起，或者其他类似的糟糕的情况，因为当试图从未预期的错误中恢复的时候现成不能执行适当的清理。
SHE要求编译器支持；在NT上，MICROSOFT C/C++支持SHE。
就象前面提到的，异常的结果是控制交给内核陷阱处理器。陷阱处理器可能处理一些明显的异常或者接着把控制交给内核中的异常分派器KiDispatchException ( )。KiDispatchException ( )接着调用RtlDispatchException（）来调用开发人员提供的可能的异常处理代码。为了异常处理器能被调用你必须使用运行时库注册。这个注册由C/C++编译器通明地为你执行，编译器会在你代码中碰到try-except结构的地方产生适当的代码来完成。同样的，自动跳出基于栈的调用帧是在代码中有try-finally结构的地方处理的，编译器和运行时库合作产生适当的代码来跳出调用帧。
这里是两个主要的SHE的结构：

try-except结构允许你创建代码来清晰地处理未预期事件或者异常，定义如下：

```
try {
// 执行代码.
```

```
} except ( /* 一个异常过滤器. */ ) {
…
// 在异常过滤器返回EXCEPTION_EXECUTE_HANDLER 的时候执行.
// 这里的代码叫异常处理器代码.
// 一旦代码执行完,控制交给try-except 结构的下一条指令.
…
}
```

这个结构由三部分组成：try结构允许你定义被异常处理器保护的代码块；except允许你指定你是否想要处理一种特定的异常；异常处理器执行和异常相关的处理。

try-finally结构允许你为一个特定的代码块指定特定的结束处理器。使用他，可以保证适当的处理总能执行，不管那种方法退出这段代码块，结构定义如下：

```
try {
//执行代码.
} finally {
// 这里执行清理.
// 写在finally结构里的代码将被执行
// 不管以什么方法退出受上面的try保护的块
// .
}
```

try-finally 结构由两部分组成：try结构允许你定义一个受结束处理器保护的代码块；finally包括组成结束处理器的代码。

try-except结构

try-except结构允许你保护一块代码，如果异常在保护代码里产生，控制将交给（用RtlDispatchException()例程）你的异常处理器，为了这个控制转移的发生，编译器和NT内核必须合作。

作为编译器，在碰到try-except结构的地方自动插入代码来为特定的代码块（叫做帧）向内核注册。就象前面描述的，内核然后就负责在异常发生的时候继续控制，接下来，内核允许你的异常处理器对异常进行一次处理。

每一个发生在受异常处理器保护的帧中的异常的引出的直接结果是最终控制交到异常处理器代码里，除非异常被调试器处理了。但是，你可能不希望处理所有可能的异常，因此你可以利用一个排除过滤器来决定你的代码是否处理这个异常。

异常过滤器是跟在except关键字后面的一段代码，注意他可以相当复杂，实际上可以调用一个函数叫做异常过滤函数来执行异常过滤的任务。他能够返回以下三个值：

```
EXCEPTION_EXECUTE_HANDLER
EXCEPTION_CONTINUE_SEARCH
EXCEPTION_CONTINUE_EXECUTION
```

EXCEPTION_EXECUTE_HANDLER返回值导致异常控制器代码被执行。在异常处理器完成他的处理后，执行流在异常处理器的下一条指令出开始。为更好的理解，考虑这个例子函数：

```
NTSTATUS MyProcedure_A {int *Somevariable}
{
    char *APtrThatWasNotInitialized = NULL;
    int Another InaneVariable = 0;
    NTSTATUS RC = STATUS_SUCCESS ;
```

```
    try {
        *SomeVariable = MyProcedure_B(APtrThatWasNotInitialized) ;
        // The following line is not executed if an exception occurred
        // in the procedure call.
        Another InaneVariable = 5;
    } except (EXCEPTION_EXECUTE_HANDLER) {
        RC = GetExceptionCode ( ) ;
        DbgPrint ( "Exception encountered with value = Ox%x\n", RC) ;
    }
    // Execution flow resumes here once the exception has been handled.
    AnotherlnaneVariable = 10;
    return (RC) ;
}
int MyProcedure_B (char *IHopeThisPtrWasInitialized)
{
    char ACharThatlWillTryToReturn = ' A 1 ;
    // Exception occurs in the following line if "IHopeThisPtrWasInitializec
    / / is invalid.
    *IHopeThisPtrWasInitialized = ACharThatlWillTryToReturn;
    // The following code is never executed if an exception occurred abov
    ACharThatlWillTryToReturn = 'B1;
    return ( 0 ) ;
}
```

正如你在代码片段中看见的，当MyProcedure_B试图用输入的字符指针参数拷贝一个字符时会发生异常。因为MyProcedure_B没有异常处理器，在调用函数中的处理器将被调用。MyProcedure_A中的异常过滤器立即返回EXCEPTION_EXECUTE_HANDLER。这导致异常处理器被调用，仅仅是取得异常代码和输出一个调试打印调用。

有趣是执行流（在异常处理器执行后）在语句AnotherlnaneVariable = 10继续，跟在导致异常的MyProcedure_B中的语句都被跳过。还有MyProcedure_A中跟在MyProcedure_B后面的语句。这里恢复执行异常处理器后面的一条指令是用跳出基于栈的调用帧来达到的。

虽然例子中的异常过滤器很没有价值，但是他确实可以很复杂。你可以调用单独的过滤器函数来决定你要处理的异常，不过记住必须返回上面列出的三个值中的一个。作为过滤函数的参数，你可以用GetExceptionCode( )内部函数调用传递异常代码或者用GetExceptionInformation()内部函数调用传递更多信息如寄存器保存的线程上下文。

GetExceptionCode( )内部函数返回异常代码，这个函数可以在异常过滤器里调用也可以在异常处理器中调用，但是GetExceptionInformation()内部函数只能在异常过滤器中调用。

通常，你的异常过滤器（或者任何过滤器函数）不需要GetExceptionInformation()内部函数返回的EXCEPTION_POINTERS结构包含的附加信息。虽然理论上可以修改这个结构中的个别寄存器在内容，这样做是极端不可移植的，可能也是不可维护的代码，我们坚决抵制这样。

注意GetExceptionCode( )和GetExceptionInformation()函数都可以在异常过滤器函数中调用。

下面的代码片段示意异常过滤器的使用：

```
NTSTATUS MyProcedure_A (int *SomeVariable)
{
    char *APtrThatWasNotInitialized = NULL;
```

```
    int Another InaneVariable = 0;
    NTSTATUS RC = STATUS_SUCCESS ;
    try {
    *SomeVariable = MyProcedure_B (APtrThatWasNotlnitialized) ;
    // The following line is not executed if an exception occurs
    //in the procedure call.
    AnotherlnaneVariable = 5;
} except (MyExceptionFilter (GetExceptionCode（）,
                GetExceptionlnf ormation ( ) ) ) {
    RC = GetExceptionCode ( ) ;
    DbgPr int ( "Exception with value = Ox%x\n", RC) ;
    // Execution flow resumes here once the exception has been handled.
    AnotherlnaneVariable = 10;
    return (RC) ;
}
int MyProcedure_B (char *IHopeThisPtrWasInitialized)
{
    char ACharThatlWillTryToReturn = 'A1 ;
    // Exception occurs in the next statement if the value of
    // IHopeThisPtrWasInitialized is invalid.
    *IHopeThisPtrWasInitialized = ACharThatlWillTryToReturn;
    // The following code is never executed if an exception occurred above.
    ACharThatlWillTryToReturn = 'B';
    return ( 0 ) ;
}
unsigned int MyExceptionFilter (
    unsigned int ExceptionCode,
    PEXCEPTION_POINTERS ExceptionPointers)
{
    // Assume we cannot handle this exception.
    unsigned int RC = EXCEPTION_CONTINUE_SEARCH;
    // This function is my exception filter function. It must return
    // one of three values viz. EXCEPTION_EXECUTE_HANDLER,
    // EXCEPTION_CONTINUE_SEARCH, or EXCEPTION_CONTINUE_EXEC0TION.
    // In our example here, we decide to handle access violations only.
    switch (ExceptionCode) {
        case STATUS_ACCESS_VIOLATION :
            RC = EXCEPTION_EXECUTE_HANDLER;
            break ;
        default:
            break;
    // If you wish, you could further analyze the exception condition by
    // examining the ExceptionPointers structure.
    ASSERT ((RC == EXCEPTION_EXECUTE_HANDLER) ||
```

```
            (RC == EXCEPTION_CONTINUE_SEARCH) | |
            (RC == EXCEPTION_CONTINUE_EXECUTION) ) ;
    return (RC) ;
}
```

异常处理器可以嵌套，任意交叉过程调用甚至在同一函数里。注意某些异常是致命错误（比如在DISPATCH LEVEL 或者更高的IRQL访问分页内存导致的页错误异常）将导致系统崩溃。不管你的代码中有没有异常处理器。因此，不要假定异常处理能保证有效的捕获所有的错误情况。

try-finally 结构：

try-finally 结构用一个“结束处理器”（*termination handler*）来保证协调地从一块代码中释放堆栈，即使发生异常导致控制突然转移到其他的调用帧上。这里的概念很简单：假设一段代码在try-finally 结构的保护之中。在控制转移到try-finally 结构结构以外的指令之前，finally部分的语句会执行。

这个简单的例子示意这种用法：

```
NTSTATUS MyProcedure_A (int *SomeVariable,int AnotherVariable)
{
    char *APtrThatWasNotInitialized = NULL;
    int Another InaneVariable = 0;
    int AnotherInaneVariable2 = 0;
    NTSTATUS RC = STATUS_SUCCESS;
    try {
        if ( ! AnotherVariable) {
            AnotherlnaneVariable = 7;
        }
        *SomeVariable = MyProcedure_B (APtrThatWasNotlnitialized,
                                       &AnotherInaneVariable) ;
        // 如果在上面的过程调用的中发生异常，下一行不会被执行
        AnotherInaneVariable2 = 5;
    } except (MyExceptionFilter (GetExceptionCode ( ) ,
        GetExceptionlnformationf ) ) ) {
        // 尽管异常到了这里， AnotherlnaneVariable 的值一定是 15
        //因为 MyProcedure_B 中的finally 部分在任何异常处理代码开始执行前执行了.
        ASSERT (AnotherlnaneVariable == 15);
        RC = GetExceptionCode ( ) ;
        DbgPrint( "Exception with value = Ox%x\n", RC) ;
    }
    //我断言  AnotherlnaneVariable 被设置为 15
    //因为赋值是在 MyProcedure_B中的finally结构中
    //因此赋值操作一定已经执行了.
    ASSERT (AnotherlnaneVariable == 15);
    // 异常处理后执行从这里重新开始.
    AnotherlnaneVariable = 10;
    return(RC) ;
```

```
}
int MyProcedure_B (char *IHopeThisPtrWasInitialized,
                   int *AnotherInaneVariable)
{
    char ACharThatlWillTryToReturn = 'A';
    try {
        if (*AnotherInaneVariable ==7) {
            // 这样做不是一件好事情.但是我们是为了这里示意
            // finally中的代码将在突然的从函数返回中得到执行的机会
            return(1);
        }
        // 如果 IHopeThisPtrWasInitialized 无效将发生异常.
        *IHopeThisPtrWasInitialized = ACharThatlWillTryToReturn;
        //如果上面的代码发生异常下面的代码决不会执行 .
        ACharThatlWillTryToReturn = 'B';
    } finally {
        //无论上面发生什么，例如，不管是执行到返回语句还是发生异常
        //下面的代码都会得到执行，注意如果发生异常
        //以下的代码将在中 MyProcedure_A 中的异常处理器得到执行之前执行.
        *AnotherInaneVariable = 15;
    }
    return ( 0 ) ;
}
```

MyExceptionFilter的代码在前面讨论try-except的时候出现过。

有三种不同的途径使控制流从MyProcedure_B中转移出去：

在try-finally结构中的返回语句没有执行也没有发生异常。

例如，如果AnotherVariable被初始化成有效的值就会发生。

这种情况下，所有在try和finally之间的语句都将执行；然后finally中的结束处理器会执行，然后通过return (0)把控制返回给MyProcedure_A。

考虑AnotherVariable值为0的情况。现在MyProcedure_B中的return (1)将把控制返回给MyProcedure_A。但是，try-finally结构中的代码会在语句return (1)处理之前得到执行。因此*AnotherInaneVariable将被设置为15。

现在假设AnotherVariable的值不为0的情况，而且AptrThatWasNotlnitialized被置为NULL.我们知道这会在MyProcedure_B中发生异常(STATUS_ACCESS_VIOLATION)。你也知道MyProcedure_A中有一个异常处理器会处理这种异常因为异常过滤器将会决定处理这种异常。

但是，在MyProcedure_A中的异常处理器得到执行之前，内核释放基于栈的调用帧。这个释放栈的过程包括执行在MyProcedure_A和MyProcedure_B的调用层次中的结束处理器保护中的所有语句。在我们的例子中，MyProcedure_A直接调用MyProcedure_B因此只有一帧要释放和一个相应的结束处理器。

因此结束处理器保护的假设的代码块（异常发生的地方）将在MyProcedure_A中的异常处理器被执行之前被执行。因此*AnotherInaneVariable将被设置为15，我们使用ASSERT来检查这个事实。

通常结束处理器不是用于处理例子中的这种简单的初始化，相反，经常用于保证在控制被交

给其他模块之前执行必要的清理操作。例如，如果分配了内存用于临时的目的，释放这个内存可以在结束处理器中来做（可以使用一些BOOLEAN变量来检查内存是否真的分配了或者如果指针总是保证被初始化为NULL的时候可以用指针的值来判断）。
相似的，如果有申请了的锁（例如一些ERESOURCE类型的资源或者一些已经申请到的MUTEX），释放可以放在结束处理器中。因此结束处理器是一种保证执行的清理申请资源的方式，不管什么方法从受保护的模块中退出。

简单的警告

在前面的例子中，我在MyProcedure_B中的结束处理器保护的代码中安排了一个return (1)。目的是演示即使是从受保护的块中这样退出结束处理器也会自动执行（由于基于栈的调用帧释放）。这使用于所有的其他C语句如break 和 continue这样导致控制被传给其他语句的语句，如果控制被传给保护帧外的其他语句，结束处理器总会先得到执行的。
但是，如果你关心你的驱动有优良的性能，你绝对不应该在结束处理器保护的帧中使用这样的语句。原因很简单：执行期间的调用帧释放是昂贵的操作。实际上，在某些处理器上，堆栈释放能在执行时候导致数百个汇编指令的执行。因此应该努力避免这种堆栈释放，除非一些异常情况发生了。

注意：异常的定义是非典型的时间，因此处理异常时候的主要的考虑是尽量文雅地恢复，处理的回报是次要的。

你可能觉得在结束处理器保护的代码中避免使用return很受限制，我同意这种看法，因此我介绍给你一种方法饶过这个限制。

```
// 谁说 goto 总是不好的????
// 定义下面的宏
#define try_return(S) { (S) ; goto try_exit; }
NTSTATUS AnotherProcedureThatUsesATerminationHandler (void)
{
    NTSTATUS RC = STATUS_SUCCESS ;
    try {
        if (!NT_SUCCESS(RC)) {
        // 假设如果上面的内存分配操作失败
        // 正常的执行不能继续.
        // 用 try_return 宏代替简单的 return 语句.
        // 注意任何合理的C语句可以在宏里执行.
        try_return(RC);
    }
    。。。
    try_exit: NOTHING;
    } finally {
        。。。
    }
    return(RC);
}
```

try_return宏简单地执行一个跳转到函数的结尾处（实际上在上面的代码中示意的是跳转到结束处理器的前面），加之，他允许你在跳转之前执行任何合法的C语句。使用try_return来代替直接使用return语句，你可以避免编译器执行调用展开（为了保证结束处理代码的执行）。

一贯地在你的代码中使用try_return宏和系统地使用异常处理器，你能完全利用结束处理器提供的威力而不用遭受与之相联系的调用展开导致的性能降低。

同时使用异常处理和结束处理

同时使用两种处理器是保护你的能处理代码未期望的异常和保证一致的清理帧的正确的方法。通常，下面的方法可以使用：

```
NTSTATUS AProcedureForDemonstrationPurposes (void)
{
    NTSTATUS RC = STATUS_SUCCESS ;
    。。。
    // The outer exception handler ensures that all exceptions will first
    // be directed to us.
    try {
        // The inner termination handler is our guarantee that we will always
        // get an opportunity to clean-up after ourselves.
        try {
            …
            try_exi t: NOTHING;
        } finally {
        // Clean-up code goes here.
        }
    } except (/* the exception filter goes here */) {
    //My exception handler goes here.
    }
    return(RC);
}
```

时间记录（Event Logging）

内核驱动常常需要向系统管理员或者用户传达信息。信息可以包括软件或者外围连接的硬件造成的，警告信息可以指示恢复错误或者可能即将发生的错误，信息（状态）消息指出一些重要的事情发生了。

NT事件日志是各个软件模块发送消息的日志的中心仓库。事件日志是一个包含已经定义好的，固定格式的事件日志的数据库。用户可以使用事件日志查看器从数据库中取出信息或者使用WIN32子系统环境提供的API来得到这些数据。

注意：虽然去解码文件中的真实的事件日志条目的记录结构这是可能的，如果你希望开发你自己的事件查看器，你可能使用基于WIN32的API去访问事件日志文件更好一些。这些API包括调用打开文件，得到一个个记录，然后关闭文件。

NT中的事件日志功能底下的概念相当简单。内核驱动记录一个事件来指示一些重要事件发生了。每个事件必须定义在消息文件中，有一个唯一的事件标志和他相关。时间标志象其他比如NTSTATUS类型的状态码一样（格式在后面说明）。记录是事件包含了信息比如事件标志，组成事件的名字，或者是与事件关联的任何的字符串或者二进制数据。事件日志还允许内核模块包括其他相关信息指出错误情况，比如操作重试了多少次，设备发生错误的偏移位置，在I/O请求包（IRP）中驱动返回给I/O管理器的状态，和其他相似的信息。

事件标志有和他们相关的替代字符串。例如，系统定义的错误 IO_ERR_PARITY有下面的代替

字符串和它相关（通常从文件*%SystemRoot%\system32\iologmsg.dll*中读取）
A parity error was detected on %l（检测到奇偶校验错误）
这个信息也示意了插入字符串的使用。插入字符串由驱动在记录事件的时候提供。第一个字符串用%1表示，第二个用%2，依次类推。驱动提供的插入字符串自动被事件查看器插入进消息文本中（它从事件记录中得到插入字符串）。
系统返回的状态码的格式和由独立驱动创建的事件标志符的格式如下：

```
31  30 29 28 27                    16 15                    0
```

S = 信息码（2位），有下面的可能值：
00 = 成功
01 = 信息
10 = 警告
11 = 错误
C = 客户定义标志（1位）。第三方定义的事件标志/状态码应该设置为1（意思是你驱动中定义的私有的状态码应该设置为1）。
R = 保留位（1位）。微软建议设置为0。
Facility（12位）。指出这个错误/状态代码所属的组，可以是下面的值：
FACILITY_NONE (0x0)
FACILITY_RPC_RUNTIME (0x2)
FACILITY_RPC_STUBS (0x3)
FACILITY_IO_ERROR_CODE (0x4)
如果开发文件系统或者过滤驱动或者其他的内核驱动，通常要为你定义的任何状态码设置Facility这个值为0x0或者0x4。
Code（16位）。可以为你的驱动中定义的每一个状态码定义任何唯一的值。你的驱动可能定义一个典型的错误代码如下：

```
#define MYDRIVER_ERROR_CODE_IO_FAILED (0xE0047801)
```

如果展开这个错误代码，我们得到二进值 1110 0000 0000 0100 0111 1000 0000 0001。
相应的这是一个错误情况（30和31位为1），是一个用户定义的错误（29位为1），保留位为1，facility说明这个错误属于FACILITY_IO_ERROR_CODE（因为facility值为0X4-16到27位），私有定义的代码是0X7801（0到15位）。注意私有定义的错误码可以是任何16位的值，只要在你的驱动中是唯一的。
因此事件查看应用程序（或者任何解释错误日志的应用程序）可以把你的事件标志关联一个文字的描述，你的驱动必须提供包含和每个事件标志关联的文本的消息文件。例如一个典型的文本消息可能这样来定义我们前面定义的错误代码：

```
The driver tried to perform an I/O operation on the device. This I/O
operation failed due to a time-out condition (device did not respond
within the specified time-out period).
```

正如你看见的，提供给用户导致失败的可能是清晰的说明是非常有帮助的。系统管理员能够根据你提供的信息开始诊断和恢复导致的原因。你的驱动使用的资源编译器可以创建适当的消息文件。应该提供什么文档给资源编译器来决定怎样创建消息文件，这是一个示例消息文件：
Sample Message File.

```
:Filename: myeventfile.mc
;Module Name: mydriver_event.h
 ； # ifndef _MYDRIVER_EVENT_H_
#define _MYDRIVER_EVENT_H_
;Notes:
; This file is generated by the MC tool from the mydriver_event.me file.
;0sed from kernel mode. Do NOT use %1 for insertion strings since
;the I/O Manager automatically inserts the driver/device name as the
; first string.
MessageIdTypedef=ULONG

SeverityNames=(Success=0x0:STATUS_SEVERITY_SUCCESS
                    Informational=0xl:STATUS_SEVERITY_INFORMATIONAL
                    Warning=0x2:STATUS_SEVERITY_WARNING
                    Error=0x3:STATUS_SEVERITY_ERROR)

FacilityNames=(10=0x004)
Messageld=0x7800 Facility=IO Severity=Informational
SymbolicName=MYDRIVER_INFO_DEBUG_SUPPORT
Language=English
This message and accompanying data is for DEBUG support only.
Messageld=0x7801 Facility=IO Severity=Error
SymbolicName=MYDRIVER_ERROR_CODE_IO_FAILED
Language=English
The driver tried to perform an I/O operation on the device. This I/O
operation failed due to a time-out condition (device did not respond
within the specified time-out period).
;Use the above entries as a template in creating your own message file.
;#endif // _MYDRIVER_EVENT_H_
```

在文本消息中包含和事件相关的插入字符串是可能的。用占位符%1，%2等表示。如果你在消息中指定占位符，驱动可以写事件日志的时候就可以提供插入的字符串。但是注意I/O管理器总是把设备/驱动名作为每一个记录的事件日志的第一个插入字符串。即使驱动提供了插入字符串，他们会被插入在设备/驱动名字后面。这样的结果就是使用%1在文本中作为插入字符串的占位符将使设备/驱动名被放入这里，而不是驱动提供的第一个插入字符串。要得到驱动提供的第一个插入字符串的话使用%2作为占位符，取第二个用%3，以次类推。

事件日志查看器怎样查找消息文件

为了事件日志查看器能找到你消息文件，你的驱动（或者更可能是安装你的驱动的应用程序）必须修改目标机器的注册表。通常，用户使用WIN32子系统作为他们的本地子系统。这种情况下，安装组件应该在注册表路径CurrentControlSet\Services\EventLog\System下面建立一个唯一的子键（有三中可能的位置Application，Security, System）。

唯一的子键应该和驱动的执行文件名字相同。例如，系统提供的AT磁盘驱动的子键是atdisk，这和驱动的名字一样（atdisk.sys）。在这个子键中，至少有两个值必须创建：

EventMessageFile

这个REG_EXPAND_SZ类型的值包含了有每个事件标志的文本消息的消息文件的完全路径和文件名。全路径名和文件名的示例可能是这样的：

%SystemRoot%\MyDriverDirectory\message.dll

TypesSupported

这个REG_DWORD值对你的驱动来说应该设置为0X7，指出你的驱动支持的事件类型EVENTLOG_ERROR_TYPE(0xl), EVENTLOG_WARNING_TYPE (0x2),和EVENTLOG_INFORMATION_TYPE (0x4)。

一旦你创建了适当的注册键，安装程序复制了适当的消息文件到相应的目录，事件查看器就能够找到和使用你的消息文件的内容。

记录事件日志条目

现在你为你的驱动定义了适当的事件标志，就可以使用I/O管理器提供的支持例程来用这些事件标志来记录事件日志。记录一个事件执行下面两步：

1， 事件/错误日志条目用loAllocateErrorLogEntry ()分配。

2， 在事件/错误日志条目初始化后，可以用loWriteErrorLogEntry()来记录。

这个函数用来分配一个事件/错误日志条目，以便后来记录，定义如下：

```
PVOID
loAllocateErrorLogEntry(
IN PVOID      IoObject,
IN UCHAR      EntrySize
）；
```

参数：

IoObject

这个参数必须或者是指向设备象，表示被记录事件/错误的设备，或者是驱动程序对象代表控制被记录事件/错误的设备的驱动程序。

EntrySize

将分配的对象的大小。因为作为开发者可以记录二进制数据，也可以提供插入数据增加你的消息，条目的大小应该这样计算：

```
sizeof(IO_ERROR_LOG_PACKET) + (n * sizeof(ULONG) +
          sizeof(InsertionStrings))
```

这里 n 等于 事件记录所要倾倒的数据的字数。

功能提供：

loAllocateErrorLogEntry函数分配一个节点给你使用。你可以初始化然后用IoWriteErrorLogEntry写入日志中。如果不能分配一个节点就返回NULL。这种情况下，你将不能写入事件，等待这个错误下次发生的时候再重试这个操作。

一个警告：这个函数引用了传进去的设备/驱动对象IoObject参数，因此一旦调用了这个函数，你必须调用函数IoWriteErrorLogEntry，这个函数会解除对象引用和释放分配事件节点的内存。

初始化错误日志很简单而且在微软的设备驱动参考中有详细的文档。

一旦得到一个事件日志，你必须调用IoWriteErrorLogEntry函数来写入事件日志，这个函数这样定义：

```
VOID
IoWriteErrorLogEntry(
    IN PVOID ElEntry;
};
```

参数：

ElEntry

初始化了的要被写入的事件记录。

功能提供：

IoWriteErrorLogEntry把要初始化的事件记录节点放到写入队列中。实际的写操作将由一个系统工作者线程异步的执行。注意虽然这个函数立即返回了，但是设备/驱动对象还没有被解除引用，将直到被异步写到事件记录里的时候才发生。

注意：事件异步写到事件记录的方式也有意思。系统工作者线程从队列中取出事件记录，插入设备/驱动名字串然后使用LPC写到一个特定的端口，那里的另一个用户空间的线程把事件节点写到事件记录文件中。系统工作者线程继续写所有的记录直到错误发生或者所有事件记录都被发送给端口句柄。

驱动同步机制

驱动程序的一个主要功能是防止数据丢失，数据丢失的一个主要原因是在两个或者更多的同时执行线程操作同样的数据结构的时候缺乏同步。因为NT可以在单处理器和对称多处理器上执行，内核驱动小心的使用同步就变的尤其重要。如果你开发设备驱动程序，当操作被中断服务例程和执行在一般IRQL层的线程共享的数据结构的时候一定要小心。

就象在第一章，WINDOWS NT系统组件 讨论的，NT内核模式环境包含NT内核执行体。执行体是抢占式的而且部分是可分页的。但是，内核既不能抢占也不能分页。虽然不能抢占，在多处理器机器上内核是可以在每个处理器上同时执行的。

在这一节里，介绍组成NT执行体一部分的驱动可以使用的各种同步原素理（primitives）。这些同步元素由NT内核或者NT执行体导出；执行体使用NT核心提供的基本元素构造更复杂的同步元素。我们的目的是介绍不同的可用的元素，解释在那里可以使用。事例代码在本书的后面里尤其是文件系统和过滤驱动的代码中有大量的某些同步元素的使用。这里提到的支持例程的调用语法的更多信息参考DDK文档。

自旋锁（spin lock）

自旋锁是提供在多处理器环境下同步的基本结构。用于相互排斥。例如，一个自旋锁用来保证只有一个执行在一个处理器上的线程能访问受他保护的共享数据。获得自旋锁后面的指令也叫临界区，临界区直到自旋锁释放的时候。

当一个处理器上的线程得到自旋锁的时候，上下文切换（线程抢占）被关闭直到这个线程释放自旋锁。同样的，任何其他的执行在其他处理器上的线程将继续试图得到自旋锁，直到成功都不会前进。这个“忙等待“（例如，不停的努力检查自旋锁是否可以使用）的方法也叫做为这个锁“自旋“，所以叫做自旋锁。

自旋的线程不能被抢占（就象得到自旋锁的线程一样），但是这些线程可以被有更高IRQL等级的中断所中断。

实际的内核使用来实现自旋锁的方法是依赖于处理器，但是，通常使用原子的“测试-设置“汇编指令来实现自旋锁。软件测试锁变量的状态，如果是空闲的，就设置为忙碌，如果锁变量忙碌，软件就重复执行”测试-设置“指令”。

注意：经常，为了减少总线的争夺，测试-设置指令操作在实现中并不是连续是使用。更适合的是，操作系统使用一次测试-设置指令，如果发现锁状态被设置为忙，然后就用平常的检测指令知道发现锁状态空闲，这时候再使用测试-设置指令去试着得到锁。

自旋锁一定会被一个处理器上执行的IRQL等级最高的线程得到，其他试图得到自旋锁的的请求也将执行。

下面的的很少的规则用来保证自旋锁的正确的行为：

一旦得到自旋锁就决不要引用任何分页数据。相似的，在得到自旋锁后执行的代码必须是非分页的。

这个限制的原因是系统不能在IRQL高于或者等于DISPATCH_LEVEL的执行中发生任何页面错误。因此，这时候发生页面错误的话，系统会检查分页的代码再调用KeBugCheck，这种情况一定是程序/设计错误的直接结果。

不要在得到自旋锁后调用其他的函数。 一定需要调用其他函数，一定保证这些函数没有引用任何分页代码或数据。

因为自旋锁必须在节点上的所有处理器上共享，尽可能少占用自旋锁的时间，然后释放了让其他处理器获得他。

有可能在设计和实现的代码中顺序的请求自旋锁，例如，你获得自旋锁#1然后又去获得自旋锁#2，等等。或者你可能实现的代码中得到一个自旋锁，然后跟着的一个或者多个调用又请求其他的同步元素（例如互斥量）。这就可能引起死锁。没有死锁检查会在一个处理器请求多个自旋锁的时候执行，因为调度和抢占会在自旋锁得到以后关掉，这很可能造成系统死锁。

NT平台有两种自旋锁：

中断自旋锁

这种自旋锁同步对设备驱动数据结构的访问。他们被那些设备驱动管理的特定设备在相关的IRQL申请和释放。设备驱动通常不自己分配自旋锁的内存，也不显式地请求和释放中断自旋锁结构。事实上，内核自动在为中断调用中断服务例程（ISR）前请求和中断相关的自旋锁。然后在执行完ISR后释放。

KeSynchronizeExecution函数可以用于同步设备驱动例程的执行和特定的中断的ISR执行。这个函数请求和这个中断指针相关的中断自旋锁，提供一个例程作为参数，为了中断再把线程的IRQL升到DIRQL。KeSynchronizeExecution然后调用指定的例程，例程的执行会和ISR同步，最后在返回用户之前释放自旋锁。

中断自旋锁在低层驱动希望同步执行一个模块和这个驱动的ISR之间必须使用。试图使用执行自旋锁将必然导致数据丢失或系统死锁发生。

执行自旋锁

执行自旋锁只能被执行在IRQL为PASSIVE-LEVEL，APC-LEVEL或DISPATCH-LEVEL的线程得到。因此通常用于高层驱动如文件系统驱动用来在多处理器环境中同步访问。当然也能够被设备驱动程序开发人员使用，只要不用来执行和ISR的同步就行了。

下面其余的讨论假设你使用执行自旋锁来同步数据访问。在这本书的后面我们还会关注执行自旋锁的用法。

要使用执行自旋锁，必须先为自旋锁结构分配足够的存储空间。自旋锁必须存储在非分页内存中。通常，你的驱动可以把自旋锁定义在驱动扩展中，驱动扩展总是从非分页内存中分配的，或者使用全局定义，因为内核驱动中的全局变量通常是不分页的；或者使用分配函数（如ExAllocatePool (NonPagedPool, sizeof (struct KSPIN_LOCK)）。

> 警告：如果你碰巧从分页池而不是非分页池中分配了一个分派器对象，你将会碰到一些位预期的系统BUGCHECK发生。你的驱动可能工作的很好。但是相关的系统将会由于在高IRQL访问分页内存异常而发生BUGCHECK。你得到的栈的轨迹可能不是指向你的驱动。这是因为内核在全局链表上存储所有的线程等待的活动的分派器对象。每个这样的链表由一个自旋锁保护。当内核得到自旋锁来遍历链表的时候，链表里的对象恰好被换出内存页面，你就会碰到系统BUGCHECK。注意对象可能你总是被换出，所以你的系统有的时候可能工作的很好，但是不是总是这样。

下面的内核提供的例程是你可以操作执行自旋锁：

KelnitializeSpinlock

这个例程接受一个分配了的自旋锁结构的指针。他会初始化这个自旋锁，而且在必须在第一次请求的到之前先调用过。

KeAcguireSpinLock() /KeAcguireSpinLockAtDpcLevel()

这些例程将不断试图得到自旋锁。想要得到的自旋锁必须作为参数传进去。任意一个函数返回的时候，自旋锁就已经得到了。这两个例程的唯一的区别是KeAcguireSpinLock()将首先把处理器的IRQL到DISPATCH LEVEL然后把就的IRQL返回给调用者，用于释放自旋锁， 而KeAcguireSpinLockAtDpcLevel()则假设当前IRQL已经是DISPATCH LEVEL。

注意：在多处理器系统中，KeAcguireSpinLockAtDpcLevel什么事也不做，例如，他立即将控制返回给调用者。因此，在多处理器上调用这个函数（如果适当的话）将得到轻微的性能上的收获。

KeReleaseSpinLock() /KeReleaseSpinLockFromDpcLevel

这些函数用于释放先前得到的自旋锁。KeReleaseSpinLock需要一个附加的参数：先前调用KeAcguireSpinLock返回的旧的IRQL。一旦自旋锁释放了，处理器回到旧的IRQL。

要求在多处理器环境同步的时候，在任意线程上下文中，处理中断中的时候，还有防止上下文切换的时候可以使用自旋锁。另外，使用自旋锁的时候前面提到的所有规则都应该遵循。但是，如果你要在多处理器环境中的一些线程上下文间同步却不关心上下文切换发生，那么其他的分派器对象（在 下一节中描述）也可以使用。

注意单词“在任意线程上下文中“（in arbitrary thread contexts）。自旋锁可以可以用于设备驱动程序，他们的入口点（比如读/写）通常自行在一些独占线程的上下文中。也可能这些设备驱动的入口点可能在高ORQL上执行。自旋锁可以自由地被这样的设备驱动使用，但是其他的分派器对象（如互斥体或者事件对象）只能在那些非任意的线程上下文中使用，也就是说，文件系统驱动或者过滤驱动使用的其他分派器对象是基于（sit above）文件系统。

注意：文件系统驱动的分派例程（例如，读/写例程）通常执行在异步操作的系统工作者线程上下文中或者由I/O请求初始化的用户线程上下文中（例如，用户调用ReadFile）。因此这是非独占的线程上下文，文件系统可以自用的等待分派器对象被置为有信号状态。

另一方面，设备驱动程序有IRP（I/O请求包）队列。设备驱动的每一个I/O请求包是依次从队列中取出来，在任意一个恰好在处理器上执行的哪个线程的上下文中。因此，因为设备驱动程序的分派器执行在任意的，未知的线程上下文中，所以是不能等待分派器对象的。线程上下文的细节在后面讨论。

分派器对象

内核分派器是一组抽象，由内核提供给执行体，用来支持同步。这些对象控制分派和同步系统操作。分派器对象可以是两种状态中的一种：

有信号状态，这时候没有线程访问受分派器对象保护的共享数据或者没有其他的线程在临界代码段中。

无信号状态，指示当前一个线程正在访问共享数据或者执行在临界代码段中。

因为你的驱动是NT执行体的一部分，所以你可以在你的驱动实现中使用这些分派器对象来同步。主要内核提供的分派器对象必须被作为透明的数据结构。内核提供所有的函数你可以用来初始化，查询状态，设置状态和清除对象的状态。你必须提供存储这些对象的存储空间。这些存储空间必须从非分页池中提供（象自旋锁一样），可以在驱动程序扩展结构中，作为全局变量，或者

动态分配的内存。

用来同步访问共享数据或者控制在临界段中的执行的方法如下：

1，线程要访问共享数据资源（例如访问一些共享数据结构或者执行临界区中的代码，因此而调用内核等待例程）

线程能够调用的等待例程是：

- `KeWaitForSingleObject()`
- `KeWaitForMultipleObjects()` 或
- `KeWaitForMutexObject()`

如果被等待的对象处于有信号状态，等待就会被满足，控制也将交给等待的线程。注意在等待被满足之前，被等待的对象将被设置为无信号状态来防止其他的可能同时调用了等待例程的线程同时也得到访问共享数据的机会。

2，在第一步中任何其他的对被设置为无信号的对象调用等待例程的线程将被挂起。

3，当第一个线程完成共享数据资源的处理，要对用来同步的对象调用一个适当的例程，KeReleaseMutex或 KeSetEvent来释放分派器对象并设置分派器对象的状态为有信号。

4，现在第一个现成释放了分派器对象，等待分派器中的一个线程得到机会访问共享数据，将会被唤醒。这个现成现在被允许访问共享数据资源。

注意：对某些同步对象，多个等待的线程会被同时唤醒。但是只有一个随后能够得到同步对象。

5，1-4会在一个线程希望访问共享数据资源的时候重复

如果线程不能访问共享数据资源（也就是，如果分派器处于无信号状态因为另一个线程正在访问共享数据资源或者在临界区中执行），线程就会被挂起或和阻塞，等待分派器对象被释放。这允许系统中的其他线程继续执行，这和自旋锁非常不一样，请求自旋锁的线程将一直处于忙等待直到得到自旋锁。因此分派器对象更加有益于系统性能。

在讨论自旋锁时提到的，驱动分派例程执行在一个任意线程上下文的不能等待分派器对象有信号。IRQL在高于PASSIVE-LEVEL执行时在非0时间间隔等待一个分派器对象被认为是一个致命错误。因此大多数设备驱动开发者不使用分派器对象来同步，但是文件系统或者过滤驱动开发者即局限于（sit above）文件系统的调用层次的可以潜在地是分派器对象。

最后，当一个线程调用内核例程去等待分派器对象（如KeWaitForSingleObject）的时候，线程可以指定超时时间。如果在超时时间间隔内分派器对象没有得到信号，线程将被用特定的状态码STATUS_TIMEOUT唤醒。这就使线程保证等待是有限的，如果时间间隔是0，线程决不会睡眠；将检查对象的状态，如果是无信号的，控制立即用状态码STATUS_TIMEOUT返回给线程。

下面的分派器对象可以用于设计和开发内核驱动：

事件（Event）

定时期（Timer）

互斥体（Mutual exclusion）

信号量（Semaphore）

除了这里列出的分派器，线程还可以等待进程，线程和文件对象结构。

事件（Event）对象

事件对象用于在多个线程之间同步执行。它记录事件的发生来决定执行流程。考虑两个线程之间的一种生产者-消费者关系：生产者线程A创建被处理的数据而消费者线程B当数据可用的时候处理数据。因为线程B不知道数据什么时候可用，他可以有两种方式来操作：

1，一直向A线程询问是否有数据可以处理。但是这样做不会有好的执行系统性能，因为昂贵的处理器循环会消耗在这种忙等待模式中。

2，等待线程A通知他数据已经可用。

因为第二中方式明显优越些，所以他常用于这种情形。可以使用事件对象来实现这种通知。（记数的信号量也可以达到这个目的）

事件对象在使用前必须初始化，开始，事件对象可以被设为无信号状态。线程B可以调用一个等待这个事件对象的调用，然后就会被挂起直到等待返回。当线程A有数据要处理的时候可以调用KeSetEvent来通知事件对象。这样就会使线程B被插入调度执行的队列中。在某个时候，系统调度器会调度线程B执行，线程B就处理数据。这个方法可以在需要的时候重复进行。

有两种类型的事件对象：

通知事件对象

在这种事件对象中，每一个等待事件对象的线程在事件对象变成有信号的时候都会被调度执行。当事件对象的状态变成有信号的时候不会自动重置为无信号状态。因此，必须有某个显式地调用KeResetEvent来把事件对象设置为无信号状态。

这种类型的事件对象通常用在这种情况下：当这个事件对象被设置为有信号状态的时候，触发所有等待这个事件的线程的执行。例如，在汽车比赛中：当出发信号发出的时候，所有比赛中的汽车都启动，试图先到达终点。

同步事件对象

这种事件对象就是我们的生产者-消费者例子。这里的事件对象被置为有信号的时候只有一个等待线程会被调度执行，事件对象也会自动被置为无信号状态。这种事件对象保证在任何时候只有一个线程访问共享数据。

下面的内核支持例程可以用来操作事件对象：

KelnitializeEvent

驱动必须从非分页池中为事件对象分配存储空间。一旦你分配了存储空间就必须在任何线程试图等待，通知，或者重置为无信号这个事件对象之前调用这个函数来初始化这个事件对象。当调用这个函数的时候可以指定这应该是一个通知型事件对象还是同步事件对象。还可以指定这个对象的初始状态是有信号还是无信号。

loCreateSynchronizationEvent

严格是说这不是一个内核支持例程，而是由I/O管理器提供的。这个函数只在NT4.0及以后的版本中有效。用这个函数可以创建或者打开一个有名的同步事件对象。因为这是有名字的，所以多个驱动就可以使用相同的事件对象来同步访问共享的数据了。

如果这个事件对象不存在，这个例程将创建这个事件对象（还要初始化为有信号状态），如果存在就打开这个有名的事件对象。下面列出的操作事件对象的函数都可以用来管理这个函数返回的事件对象指针。当你的驱动不再需要使用这个事件对象的时候应该调用ZwClose例程来关闭返回的句柄。

KeSetEvent

这个例程允许你把事件对象的状态设置为有信号。一个或者多个等待这个对象的线程将被调度执行。考虑下面的伪代码片段：

```
thread_A {
    while (TRUE) {
        create new data;
        signal event object 1;
        wait for event object 2 to be signaled;
    }
}
thread_B {
```

```
    while (TRUE) {
        wait for event object 1 to be signaled;
        process data;
        signal event object 2;
    }
}
```

代码描述了一个典型的生产者-消费者关系。我们在这里看见每个线程通知一个对象后立即执行一个等待操作。

通知一个事件对象是在系统调度器可能执行一个上下问切换的时候。但是，因为我们的线程将在通知操作后自动进行睡眠，他们将被调度出去，然后在某个时候将被重新调度然后又立即进入睡眠中，这看起来相当浪费。相反，如果我们只允许在通知操作后执行那么他们就能一直睡眠而避免了一次不必要的上下文切换。这可以在调用KeSetEvent的时候指定等待参数为TRUE来完成。

注意：实现POSIX风格线程的条件变量要求原子地释放互斥对象然后使释放互斥对象的线程睡眠。这可以简单地在调用KeSetEvent的时候指定等待参数为TRUE来完成。

`KeResetEvent /KeClearEvent`

这两个函数允许你把对象的状态设置为无信号，`KeResetEvent`还返回原来的状态。

KeReadStateEvent

这个函数得到事件对象的当前状态。

定时器对象

定时器对象用于记录经过的时间。如果一个线程想在一段时间间隔后或者在特定时间执行一个任务，应该使用定时器对象。定时器对象有一个相关的状态，有信号或者无信号。当期望的时间间隔经过了，定时器对象被设置为有信号，所有等待这个定时器对象的线程的等待被满足，将会被调度执行。

象其他分派器对象一样，驱动必须从非分页池中分配定时器的存储空间。定时器对象必须初始化为无信号状态。

有两种方式在你的驱动中使用定时器对象：

1，驱动中的一个现成可能初始化一个定时器然后调用等待例程（如KeWaitForSingleObject）来挂起，直到定时器对象被设置为有信号状态（指定是时间间隔经过了）。

2，在设置定时器在一段时间间隔后终止的时候可以指定一个延迟过程调用（DPC）。当定时器时间间隔终止的时候，DPC例程将被调度执行，所有的处理请求就可以在DPC例程中执行。

注意：DPC是另一个影响内核操作的途径。DPC有进入当前运行线程中执行的能力（通过软件中断）。执行一个IRQL在DISPATCH-LEVEL的特定的过程。在执行DPC过程的时候不能调用系统服务，也不能发生页错误。另外，DPC不象APC那样针对特定的线程。当当前的IRQL低于DISPATCH-LEVEL的时候就会发生一个软件中断，DPC分派器将被调用。通常，DPC被设备驱动程序用来完成中断处理相关的处理。

在单处理器上，在任何时候只有一个DPC执行。但是在多处理器上可能在每一个处理器上都有一个DPC同时在执行。当DPC在DISPATCH-LEVEL上执行的时候处理器上的线程调度将挂起。

WINDOWS NT4.0中有两种类型的定时器对象：

通知型定时器

当这种定时器有信号的时候，所以等待这个定时器对象的线程的等待都被满足。这些线程

都得到调度执行的权利。

同步定时器

当这种类型的定时期有信号的时候，只有一个等待的线程的等待被满足。定时器对象自动被设置为无信号状态。

WINDOWS NT4.0对定时器做的另一个改进是现在可以指定周期性的定时器。这些定时器被自动插入周期一样的活动定时器列表中。

下列的内核模式可以用来操作定时器对象：

`KelnitializeTimer/KeInitializeTimeEx`

这个例程的第二个版本只能在WINDOWS NT4.0及以后版本中。这个例程要求一个在非分页池中分配的定时器对象的指针。他会初始化定时器的状态为无信号。

`KeInitializeTimeEx`函数可以指定定时器的类型（通知型或同步型）。

KeSetTimer/KeSetTimerEx

这个函数允许你设置定时器对象。时间值是用系统时间单位来表示的（100纳秒）。你有两个选择：如果指定值为负值，这个值将被解释为调用这个例程时的当前时间。如果是正值就被解释为一个绝对值；系统启动时的时间是0；

KeSetTimerEx只能在WINDOWS NT4.0及以后版本中使用，你可以指定希望定时器被激活的次数。

记住你可以指定定时器有信号时候可以调用的DPC例程。

KeReadStateTimer

得到定时器的当前状态。

KeCancelTimer

这个函数取消一个先前设定的定时器（如果还没有终止），如果定时器有相联系的DPC例程也会被取消。

这里要注意两点，首先 ，取消定时器不会设置定时器为有信号。其次，如果定时器预先已经终止而且相关的DPC例程已经在队列中，这个DPC就不会被取消。只有在定时器没有终止，相关的DPC例程没有进入队列的情况。

互斥体对象

互斥体对象和自旋锁相似，他们都只允许一个线程在任何时候访问共享数据。其他的试图得到相同互斥体对象的线程将被挂起直到第一个线程释放了互斥体对象。它和自旋锁之间的区别是等待的互斥体的线程将被挂起。

存储互斥体对象的空间必须由驱动从非分页池中提供。驱动还必须保证得到互斥体对象后的代码不会被分页。

互斥体对象有两种：

快速互斥体对象

快速互斥体对象只是事件分派器对象的一个简单的包装。它用在任何时候只允许一个线程得到斥体对象来提供互相排斥的语义学。当互斥体对象释放的时候（相应的事件对象有信号），只有一个等待的线程会被调度执行。因此，快速互斥体对象的概念和同步事件对象是一样的。

快速互斥体对象不提供任何死锁防止的支持。快速互斥体对象也不能递归地请求。因此，如果你在代码实现中有一个线程试图得到#1快速互斥体对象然后得到#2快速互斥体对象，但是另一个线程以相反的顺序做。那么就会发生死锁。类似的，任何线程试图递归的得到一个快速互斥体对象就会自己死锁。

快速互斥体对象是由NT执行体支持的，因为快速互斥体对象不是属于NT内核导出的原始的

同步机制。使用快速互斥体对象比使用NT内核支持的普通互斥体对象更快。下面的例程用于管理快速互斥体对象：

`ExInitializeFastMutex`

初始化传进去的快速互斥体对象结构。这实际上是初始化组成快速互斥体对象的事件对象的个宏。

`ExAcquireFastMutex/ExAcquireFastMutexUnsafe`

如果快速互斥体对象当前没有被其他的线程得到，这个函数将得到快速互斥体对象。随后的其他的试图得到这个互斥体的线程将被挂起直到互斥体被释放。

如果互斥体已经被其他的现成得到，当前线程将被阻塞直到互斥体可用。

这两个函数的区别很简单：如果使用`ExAcquireFastMutex`，执行体关闭分发`APC`给得到快速互斥体的线程。如果使用`ExAcquireFastMutexUnsafe`，执行体假定调用是在一个临界区中保护着的，而不用费心去关闭`APC`。

注：高层的驱动如文件系统驱动可以调用KeEnterCriticalRegion和 KeLeaveCriticalRegion 来通知当前线程进入或者离开临界区。调用KeEnterCriticalRegion关闭内核模式的APC。

KeLeaveCriticalRegion重新打开调用线程的内核模式的APC。当你的驱动觉得在他处理的时候不想接收核模式的APC的时候在应该调用 KeEnterCriticalRegion。

注意：APC是一种能影响线程的控制流的方法。一个APC必须针对一个特定的目标线程。这是和DPC的区别，DPC执行在当前在处理器上执行的任意线程的上下文中。APC针对的线程将被中断（通过软件中断）。在创建APC的时候指定的过程将被在被中断的线程的IRQL为APC-LEVEL等级执行。

APC在用户模式和内核模式都可以分发。内核模式APC有两种：普通的和特殊的。普通的内核APC可以被驱动调用KeEnterCriticalRegion来关掉。但是特殊的APC不能关掉。参考DDK得到更多的APC的信息。

ExReleaseFastMutex/ExReleaseFastMutexUnsafe

释放先前得到的快速互斥体。分别对应于调用`ExAcquireFastMutex` 和`ExAcquireFastMutexUnsafe`来得到的互斥体。

ExTryToAcguireFastMutex

这个函数试图得到快速互斥体。如果成功返回TRUE（将阻塞内核模式的APC）。如果不能得到快速互斥体返回FALSE。调用者接着可以立即重新试或者一段时期后再试。

互斥体

互斥体类似与快速互斥体。但是互斥体是由NT内核提供的，他有下面的快速互斥体没有的特征：

1，驱动可以在初始化每个互斥体的时候关联一个等级。

内核检查得到的互斥体的等级来保证先前得到的互斥体的等级都比当前互斥体的等级底（除非是递归得到的同一个互斥体）。

注：互斥体相关的等级应该和你驱动的锁定的层次相符合。例如，如果你的锁定层次指出互斥体#1总是在互斥体#2之前得到，那么你就应该给互斥体#1关联一个较底的等级（低于0的值）而给互斥体#2关联一个较高的值（大于0的值）。

2，互斥体可以递归得到。

因此，你驱动中的线程就能够安全的重新得到同一个互斥体多次。唯一的限制是这个互斥体应该被释放的次数和得到的次数一样。

3，当你驱动中的线程等待的互斥体对象得到的时候，这个线程的优先级升高到适时优先级。这个优先级在释放互斥体对象的时候自动降低。

4，所属进程（得到互斥体的线程所在的）不能被换出内存。

下面是NT内核提供的支持互斥体对象的例程：

KelnitializeMutex

驱动必须指定一个有效的非0等级参数，如果需要并发的得到多个互斥体对象（如果指定0作为每个互斥体对象的等级值，当你试图并发的请求多个互斥体对象的时候将导致BUGCHECK）。

KeReadStateMutex

返回互斥体对象的当前状态。

KeReleaseMutex

释放先前得到的互斥体对象。如果释放互斥体对象的线程希望立即执行内核等待例程（例如，KeWaitForSingleObject），应该指定等待参数为TRUE，避免不必要的上下文切换。

信号量对象

信号量对象（记数信号量对象）允许一个或者多个线程同时访问共享数据资源。可以指定只允许一个线程访问数据资源来提供互相排斥（和互斥体类似）。通过灵活地指定能够访问共享数据的线程的数量，是控制平行的访问的理想方法。信号量对象应该被看成门。当没打开的时候，同时访问共享数据资源是允许的。一旦门关上了，就没有更多线程能访问共享数据资源了。

虽然和互斥体类似，但是信号量对象不提供互斥体提供的死锁检测功能。得到信号量对象也不会引起内核APC关闭。注意信号量对象的存储空间必须从非分页内存中分配。

这里是信号量对象的工作方式：每个信号量对象有一个相关的记数值。如果这个相关的记数值是0，任何等待这个信号量对象的线程都会被挂起。当得到这个信号量对象的线程释放信号量对象的时候，计数器值增加一定数量（在释放的时候由Adjustment参数指定）。如果增加了的计数器的值在0到一个非0值的范围中，那么某几个等待的线程的等待就会结束。

每次一个等待结束计数器会减一；因此，和计数器的值相等的数量的线程的等待将得到满足。接下来的结果是一定数量的（在初始化信号量的时候指定的限制值是上限）线程能够同时得到信号量对象来访问共享数据资源。

下面的例程有NT内核提供来支持记数信号量对象：

KelnitializeSemaphore

可以指定相关计数器的初始值，如果计数器的值为非0，信号量将被设置为有信号状态。你必须指定信号量允许的最大值。这个限制参数限制能够同时访问信号量保护的共享资源的数量。

KeReleaseSemaphore

在释放一个信号量对象的时候，你的驱动可以指定和信号量相关的计数器应该增加的值。这可以使一个或多个等待的线程的等待结束。注意，如果这个指定的计数器要增加的值最后导致计数器的值超过原来的上限值（在初始化的时候指定的），或者你指定一个负值，执行线程将会发生一个STATUS_SEMAPHORE_LIMIT_EXCEEDED异常。

KeReadStateSemaphore

这个例程返回和信号量对象相关的计数器的当前值。这个值可以被看作马上要结束的等待的数量。

资源对象（ERESOURCE）（读/写锁）

WINDOWS NT执行体提供一个重要的附加的同步机制广泛地用于文件系统驱动，ERESOURCE是一个提供单个的写（互斥访问），多个读（共享访问）语义的基本的结构对象。因此每个线程能够灵活的决定请求资源对象的访问类型。

当一个线程要修改资源对象保护的共享数据的时候，他必须互斥的请求读写锁。但是如果现成仅

仅是读资源对象保护的共享数据，他通常请求共享的资源对象，允许其他线程同时读相同的共享数据。当然，如果任何线程得到互斥资源，就没有其他的线程能得到。

存储读写锁的空间必须由驱动程序从非分页池中分配。

资源结构对资源来说有所属线程的概念（多个读线程同时属于共享的资源）。另外，这些读写锁能够被递归地获得。但是线程必须释放他得到一样多的次数。

注意：在这一章讨论的原始的同步分派器中，没有分派器对象在驱动发现不再需要而要释放同步对象占用的内存的时候需要反初始化的。但是，资源（ERESOURCE）结构必须在释放结构的内存之前反初始化（或者从资源结构的全局链表中删除）。

最后，所有资源操作例程要求处理器的IRQL小于或者等于DISPATCH-LEVEL。

注意：ERESOURCE结构使用执行自旋锁来保护在资源结构内部的内部域。当得到了这个自旋锁的时候，执行体会把处理器的IRQL升高至DIAPATCH-LEVEL。因此，调用任何IRQL高于DISPATCH-LEVEL的例程将导致死锁。

下面的NT执行体提供的例程支持ERESOURCE结构：

ExInitializeResourceLite

这个例程初始化驱动分配的资源结构。资源被插入资源结构的全局链表中，因此，在释放分配的内存之前反初始化是很重要的。

ExDeleteResourceLite

这个例程把资源从全局资源链表中脱链。为资源分配的内存可以在随后释放。

ExAcquireResourceExclusiveLite

这个例程试图得到互斥资源结构（写访问）。请求互斥访问的线程可以指定是否希望阻塞等待资源变得可用为止。如果线程不准备阻塞，而且其他线程已经得到了共享的或者互斥的资源结构，这个函数会返回FALSE，指出请求不成功。

ExTryToAcquireResourceExclusiveLite

这个函数功能上等于调用ExAcquireResourceExclusiveLite而把等待（Wait）参数设置为FALSE。但是，MICROSOFT声称这个函数更有效率。

ExAcquireResourceSharedLite

这个例程试图共享的（读访问）得到资源结构。请求互斥访问的线程可以指定是否希望阻塞等待资源变得可用为止。如果线程不准备阻塞，而且其他线程已经互斥的得到了资源结构，这个函数会返回FALSE，指出请求不成功。如果其他线程共享的得到了资源结构，当前的共享访问请求将成功然后返回TRUE。

ExReleaseResourceForThreadLite

调用这个例程释放先前得到的资源结构。线程ID（标志执行这个操作的线程）必须作为参数传递。线程ID可以调用ExGetCurrentResourceThread得到。

ExAcquireSharedStarveExclusive

通常，请求资源结构是可管理的因此线程请求互斥访问不会饿死，饥饿会在下面的情况下发生：

一个线程已经得到共享的资源结构。随后，一个请求互斥的资源结构的请求到来而且等待参数设置为TRUE。这个请求因此被排队。在有共享资源结构的线程释放资源结构之前，其他的请求共享资源的请求达到。如果NT执行体使共享的请求得到满足而使请求互斥的访问的请求等待，就可能请求互斥的访问的请求将会饥饿（即是说，永远无法完成）。

因此，NT执行体通常不会在已经有一个请求互斥的访问资源的请求的时候满足新的共享访问请求。注：如果一个已经拥有互斥访问资源结构的线程又请求共享资源访问结构（递归请求），这个请求总是会被同意。

但是使用这个调用，线程故意请求共享访问使优先于任何先前的请求互斥访问的队列。

`ExAcquireSharedWaitForExclusive`

这个例程是前一个例程的相反方面，这里，一个明确的共享访问请求声明互斥访问请求应该被优先处理，即使这个互斥访问请求后到达。因此，当前请求只有在没有未决的互斥访问资源的请求才能被满足（除非是这是一个递归请求）。

运行时支持例程（RTL）

WINDOWS NT执行体通过运行时库和文件系统运行时库向内核模式驱动开发者提供了丰富的支持。如果你准备开发内核模式驱动程序，这些库应该被仔细地了解。

运行时库由下面的例程集组成：

操作双链表

查询和写入NT注册表

执行类型转换例程（字符到串等）

执行ASCII和UNICODE字符串的操作（包括从ASCII转换到UNICODE等）

复制，清0，移动，填充和比较内存块

执行32位整形数运算和64大整数和长整数的运算（包括类型间的转换）

执行时间操作和转换的例程

创建和操作安全描述符

虽然这里没有详细讨论包含在这两个库中的例程（第二章，文件系统驱动开发，有一些其中的讨论），遍及本书的示例代码会用到一个或者更多的包含在那些库中的函数，结构和宏。

运行时库函数都以Rtl为前缀，而文件系统运行时库函数都以FsRtl为前缀。很容易区别他们。

我高度建议你熟悉这两个库提供的功能，然后在你需要在驱动中使用的时候使用这些例程。在你使用标准C库例程的时候你应该使用运行时库例程，例如，用支持例程RtlCopyMemory代替memcpy库调用，这会保证你的驱动在所用的平台上都有正确的行为。

虽然这两个库的头文件必须从MICROSOFT的IFS开发包中购买，本书将示例这些库提供的重要结构定义和函数声明的描述和使用例子。