In this chapter:
- *Exception* [illegible]
- *Structured Exception Handling (SEH)*
- *Event Logging*
- *Driver Synchronization Mechanisms*
- *Supporting Routines (RTLs)*

3

Structured Driver Development

Writing a kernel-mode driver is not easy. Unfortunately, installing a new kernel-mode driver on your production system is sometimes even worse. Drivers that execute as part of the NT Executive could potentially crash your system and do so in a way that makes it extremely hard to identify the responsible module. Furthermore, system crashes could occur with a certain regularity, or they might occur only occasionally (typically, it seems, when you are praying hard that they do not occur because you are doing something extremely important). Worse, a kernel-mode driver could corrupt your data, and do so in such a way that by the time you discover the corruption is taking place, it is too late to recover your data.

Therefore, if you are installing a new kernel-mode driver, there are certain expectations that you would have from such a driver, such as:

- The driver should not cause data corruption. This is a fundamental responsibility for kernel-mode driver designers and developers, and unfortunately, it is the hardest characteristic to evaluate objectively.*
- The driver should not cause system crashes. The objective is to ensure that even under adverse circumstances, when externally connected devices (such as disk drives or network cards) are not functioning correctly, the drivers must manage such errors gracefully. Note that the definition of handling an error gracefully is slightly nebulous: it might mean that the driver should be able to work around the situation if possible, or it might mean that the driver should (at the very least) be able to shut itself down (i.e., not provide the

---

* Therefore, it's rare that good system administrators take new drivers and install them in production environments immediately. A wise alternative would be to try out the driver on noncritical machines and evaluate its behavior over a reasonably long period of time. Unfortunately, the trial environment will probably not be an exact duplicate of the environment on the production machines, and the system administrator still can't be certain that the driver won't corrupt data in high-load, production environments.

associated functionality), but still allow the rest of the system to continue functioning normally.

- Expanding on the preceding point, software errors present in the driver code (which inevitably occur even when exceptional care is taken by developers) should not cause the system to crash. This might seem paradoxical since *bugs,* by definition, are unexpected and hence difficult to predict and manage. However, in many cases it is possible for kernel-mode driver developers to prepare for the eventuality that software errors might creep into the code and might not all be discovered during in-house testing. If the resulting driver is implemented correctly, it is indeed possible to ensure that, in most cases, such bugs do not result in system crashes.
- The driver should be able to provide adequate status reports to the system administrator. For example, if an error condition occurs, a clear, concise description of the problem should be conveyed by the driver to the administrator, allowing the administrator to try to rectify the problem if possible, or to be aware that certain loss of functionality has either already occurred or might occur shortly. Even when the driver and its devices are functioning correctly, there might be situations in which clear, concise status reports should be provided to system administrators. Data provided by drivers during error conditions could include information on recovered errors, certain performance-related statistics, or the values of automatically tuned driver parameters. This would allow administrators to understand the behavior and limitations of the system and might also afford them the opportunity to modify the work load or to further fine-tune driver parameters based on expected usage patterns.

The responsibility of achieving the expectations of the user falls squarely upon the kernel-mode driver designers and developers. The good news, however, is that it is possible to develop software for the Windows NT platform that meets the expectations listed here; indeed, the operating system provides ample support for developers to allow them to incorporate such desired features into their drivers.

Exception Dispatching Support

An exception is an atypical event that occurs due to the execution of some instruction by a thread. Exceptions are processed synchronously in the context of the thread that caused the exception condition. Since exception conditions are synchronous events that occur as a direct result of the execution of an instruction, they can be reproduced, provided that exactly the same conditions can be regenerated and the instruction is retried. The Windows NT Kernel provides support for exception dispatching when an exception is encountered.

1. Creates a trap frame for the thread. This trap frame contains the contents of all the volatile registers, i.e., registers with contents that might get overwritten as a result of processing the exception condition.
2. Optionally creates an exception frame, which contains the contents of other nonvolatile registers. The trap handler module always creates an exception frame when processing an exception condition.
3. Creates an exception record structure, which contains the exception code describing the exception that occurred, the exception flags, the address in the code at which the exception occurred, and any other parameters that might be associated with the specific exception condition. The only value that is currently legal for the exception flags field is EXCEPTION_NONCONTINUABLE, indicating that this is a fatal exception and further processing should be terminated.

   Some exceptions may have additional parameters that are supplied by the trap handler to provide more information about the exception condition. The only such exception condition that has additional parameters supplied is EXCEPTION_ACCESS_VIOLATION, which provides two associated arguments, one indicating whether it was a read or a write operation that caused the access violation, and the other is the virtual address that was inaccessible.
4. Transfers control to the Windows NT kernel exception dispatcher module.* The exception dispatcher module in the Windows NT Kernel is called KiDispatchException().

Possible Outcomes from Processing an Exception

The exception dispatcher module in the kernel determines the processor mode in which the exception condition occurred. User-mode exceptions and kernel-mode exceptions are handled slightly differently, but the basic philosophy is the same. Before we go through the steps that the exception handler undertakes in processing the exception condition, we will first discuss briefly the possible outcomes from processing an exception condition.

Each exception condition can be processed by the exception handler module in one of three ways:

- The exception handler changes one or more of the conditions that caused the problem and then directs the exception dispatcher to retry the instruction.

---

* Some exception conditions are automatically handled by the trap handler and do not require transfer of control to the exception dispatcher module. For example, debugger breakpoints are handled by directly invoking the debugger.

For example, consider an exception that indicates a page fault occurred. The exception handler will invoke the page fault handler to bring the contents of the page into system memory from some secondary storage device or from across a network. The memory access is then retried and should now succeed.

Another example is when a code segment tries to allocate some memory and subsequently tries to access it. If the original memory allocation failed, accessing a pointer to that memory block results in a memory access violation. The following example illustrates such a condition:

```
int    *SomePtr = NULL;

// 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.
```

Invoking RtlZeroMemory() with a NULL pointer results in the EXCEPTION_ACCESS_VTOLATION exception being raised. In this case, the exception handler can set the value of SomePtr to point to some preallocated memory and retry the instruction.

WARNING Retrying the assembly code that corresponds to RtlZeroMemory() (in this case) could lead to unexpected error conditions. Unless the compiled assembly code is closely examined, you can't be sure the modification of SomePtr in the exception handler results in the expected behavior, e.g., the compiler might have initialized a register with the initial value of SomePtr, which was NULL. Now, the value contained in the register will always be reused because the instruction that was retried might be a *move memory* instruction following the one that initialized the register. This means the reinitialization of SomePtr in the exception handler won't take effect, and the exception condition simply reoccurs in an infinite loop. Therefore, retrying of instructions that have led to an exception condition is a tricky proposition at best.

If desired, exception handlers can return a specific return code, indicating that the instruction resulting in the exception condition should be retried. Note that the constant value is actually unimportant but the fact that such a value is returned is noteworthy.

- The exception handler decides that the exception condition is one it does not wish to process.

  If this happens, an appropriate value is returned indicating that the exception should be propagated. Therefore, the exception dispatcher will continue searching for other handlers that might wish to process this exception.

  For example, imagine that a particular exception handler can process only one type of exception condition, say EXCEPTION_ACCESS_VIOLATION. If an exception indicating data misalignment is encountered, this particular exception handler will return a value indicating that it could not process the specific exception condition and that the dispatcher should continue traversing the call frames, looking for an exception handler that is prepared to process this specific exception.

- The exception handler executes a specific block of code, which processes the exception condition and then indicates to the caller that execution should resume following the exception handler code.

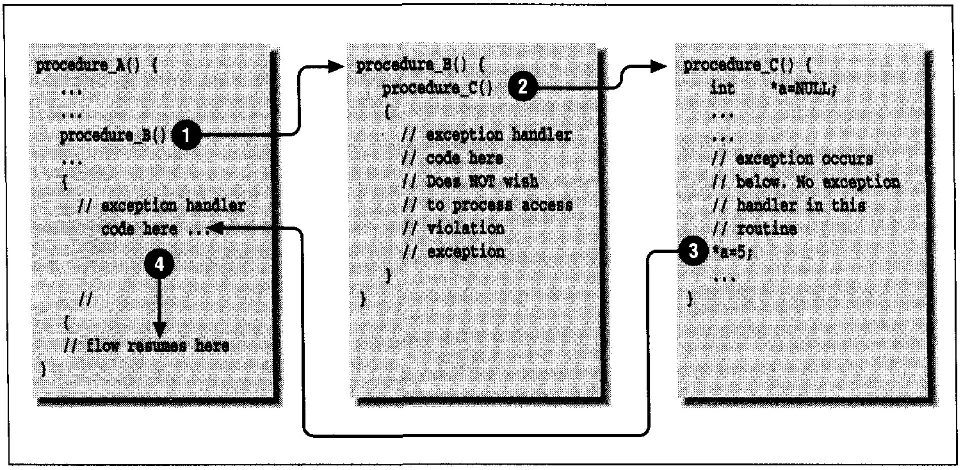
  In this case, the exception handler does not retry the instruction that caused the original exception condition, but instead tries to resume execution at the instruction immediately following the exception handler code.

  This method of processing the exception condition is substantially different from simply modifying some condition and retrying the instruction as described previously. Here, the exception handler performs some processing and then wants the instruction execution to resume immediately following the exception handler code. In the case where some condition causing the exception was modified (described earlier), the exception handler wanted execution to commence at the same instruction that caused the exception condition in the first place.

It is not necessary that the exception handler reside in the same function or procedure in which the exception condition occurred. It could have been in any of the routines that comprised the calling hierarchy.

For example, consider a situation in which *procedure_A* invokes *procedure_B,* which in turn invokes *procedure_C.* Further, imagine that an exception condition (say EXCEPTION_ACCESS_VIOLATION) was encountered at an instruction in *procedure_C.* It is quite possible that neither *procedure__B* nor *procedurejC* have an exception handler that is prepared to process the exception condition.

If an exception handler resides in *procedure_A*, and if the exception handler in *procedure_A* handles the exception condition, execution flow will resume in *procedure_A* at the instruction immediately following the exception handler code. The stack frames for *procedure_C* and *procedure_B* will be automatically unwound in order to resume execution in *procedure_A*. Figure 3-1 illustrates this.



```
procedure_A() {
  ...
  ...
  procedure_B()
  ...
  {
    // exception handler
       code here ...

    //
  {
  // flow resumes here
}
```

```
procedure_B() {
   procedure_C()
   {
     // exception handler
     // code here
     // Does NOT wish
     // to process access
     // violation
     // exception
   }
}
```

```
procedure_C() {
   int    *a=NULL;
   ...
   ...
   // exception occurs
   // below. No exception
   // handler in this
   // routine
   *a=5;
   ...
}
```

Figure 3-1- Flow of execution when an exception handler handles an exception

As you see, there are three different ways in which an exception handler might respond if called upon to process a particular exception condition.

As will be discussed later, part of unwinding the stack frames for *procedure_C* and *procedure_B* will cause any appropriate *termination handlers* to be invoked. This allows for systematic unwinding in those procedures and thereby prevents nasty side-effects such as deadlock conditions which might otherwise occur due to this unexpected transfer of flow of control from *procedure_C* to *procedure_A*.

Dispatching Kernel-Mode Exceptions

Given these different ways in which an exception handler might process an exception condition, it is useful to understand what the exception dispatcher code in the NT kernel, KiDispatchException(), actually does to process the exception condition. The following sequence of steps listed is executed by KiDispatchException() when it's invoked for an exception condition caused by a thread executing kernel-mode code:

1. First, the exception dispatcher checks to see if a debugger is active, and if so, it transfers control to the debugger.

   The debugger will indicate either that the exception has been processed by returning TRUE, or that the exception was not processed and the search for another handler should proceed.

   Note that the debugger may have modified the current instruction pointer and/or the current stack pointer obtained from the execution context structure passed to it. Therefore, if the exception is processed by the debugger, execution will not necessarily resume at the same instruction that caused the exception condition.

   If the debugger returns TRUE, indicating that it has processed the exception, KiDispatchException() returns control back to the thread that caused the exception via the trap handler, which was responsible for invoking the dispatcher routine.

2. If a debugger is not present or if the debugger returns FALSE, indicating that it did not process the exception, the dispatcher attempts to invoke any call frame-based exception handlers.

   Invocation of a call frame-based exception handler is performed via an RTL function call, RtlDispatchException(). This routine is not typically exposed to third-party driver developers.

   The RtlDispatchException() function searches backward through the stack-based call frames looking for an exception handler prepared to process the exception. This search continues until either some exception handler returns a code indicating that the instruction that caused the exception condition should be retried, or the entire call hierarchy has been examined for possible appropriate exception handlers and none were found.

   Compilers for the Windows NT platform that support structured exception handling register exception handlers with the RTL, on behalf of executing threads. Unfortunately, the functions used to register and deregister exception handlers are not ordinarily exposed to third-party developers. The good news, however, is that it would be rare for a kernel-mode driver to need to access these routines directly, since the compiler typically provides structured handling support and performs the dirty work for you.

   The RTL package also provides an interface for compiler developers to register termination handlers on behalf of executing threads. Termination handlers are described later in this chapter.

3. In Step 2, a call frame-based exception handler may or may not be found.

   Even if one or more exception handlers were identified, these exception handlers might not be prepared to process the specific exception condition.

> As described later in this section, structured exception handling allows you to check the type of exception condition and determine whether you wish to handle the exception in your exception handler or whether you wish to propagate the exception condition to the next possible exception handler.
>
> If the return code from RtlDispatchException() is FALSE, indicating that the exception has not been processed, the exception dispatcher once again attempts to invoke any debugger that may be executing. This is called *second chance* processing and the actions undertaken at this time are the same as would be taken had a recursive exception been encountered.
>
> If a debugger is connected, it has a final opportunity to keep the system alive. However, if no debugger is connected or if the debugger once again returns FALSE, the exception dispatcher will invoke KeBugCheck() and halt the entire system, resulting in the dreaded blue screen of death.
>
> The reason for the system crash will be given as KMODE_EXCEPTION_NOT_HANDLED. This indicates that no exception handler was found that would process the exception that occurred during kernel-mode execution. Unless extreme measures are called for, your code (if you happen to write a kernel-mode driver) should never, ever, cause such a blue screen.

Exception dispatching support, as well as support for registering call frame-based exception handlers, is provided by the Windows NT operating system and is not a function of (or dependent upon) any specific compiler. That said, you might note that unless you use a compiler that supports and uses the Windows NT exception handling model, you cannot develop code that can take advantage of the exception handling features provided by the operating system. The Microsoft C/C++ compiler provides structured exception handling support to both user-mode and kernel-mode code.

The Exception Dispatcher: User Mode Exceptions

The direct invocation of RtlDispatchException() by KiDispatchException() is done only for exceptions that occur while code is executing in kernel mode. If the exception occurs in user mode, KiDispatchException() performs slightly different processing.

A message is sent to the process's debug port using LPC (the local procedure call interface). If the port processes the exception, there is no further work for KiDispatchException().

Consider the case, however, where the debug port for the process fails to handle the exception. Attempting to execute user-mode exception handlers in the kernel would not be a wise thing to do. At the very least, it would introduce a large security hole in the operating system. Therefore, the KiDispatchException()

function prepares to transfer control to a corresponding user-space excepi dispatcher module.

The dispatcher function pushes the trap frame, the exception frame, and exception record on the user space stack. Then, KiDispatchExecutioi modifies the exception record such that, once control is returned from the exc tion dispatcher and the trap handler, a special user-space routine will be invo that will further process the exception condition by invoking any user-mi exception handlers. Note that the modification performed here involves chang the instruction pointer value in the exception record to point to the user-sp exception dispatcher function.

Treatment of user-mode exception handlers is similar in that the calling hierar is examined to see if any exception handler can be found that is preparec process the exception. If none of the user-mode exception handlers process exception condition, the process containing the thread that caused the excep condition is typically terminated. Termination of the user-space thread is genet done by the default exception handler, which is usually installed by the Wi (and other) subsystems.

Now that you understand the sequence of actions that take place when an exc tion condition occurs, the next logical question to ask is: How do I write c that would be able to process unexpected events, such as exception conditi Good question, and the next section provides you with one answer.

Structured Exception Handling (SEH)

The Windows NT Executive makes extensive use of structured excep handling. Each of the NT kernel-mode components tries to prepare for the eve ality that unexpected error situations might occur as a result of executing a and these modules work hard to ensure that any unexpected error conditions not bring down the entire system. It is extremely good practice for indepen driver developers to also implement structured exception handling in their dr implementations. This results in more robust kernel-mode drivers, leading t more stable Windows NT system, which in turn results in happier customers.

TIP As a kernel-mode driver designer, you can choose to avoid using structured exception handling in your driver. Many kernel-mode drivers do this and get away with it. However, if you develop file system drivers, I strongly urge you to use SEH in your driver, not only because it's the right thing to do, but also because some of the Windows NT Cache Manager support routines and some of the Virtual Memory Manager functions will raise exceptions, instead of returning errors under certain conditions that aren't catastrophic and shouldn't result in a system panic. The expectation is that the file system driver will handle such exceptions and treat them as regular error conditions.

If your driver uses structured exception handling, you can handle such exceptions gracefully; failure to use structured exception handling will result in an otherwise avoidable bugcheck condition.

Before we discuss what structured exception handling (SEH) is and the benefits that SEH can provide, let me tell you what structured exception handling cannot provide.

Structured exception handling is not a panacea for bad driver design or shoddy implementation. If you do not take care during driver design and development, no amount of structured exception handling is going to rectify the situation. Similarly, SEH cannot ensure that system crashes can be completely avoided; trying to access paged memory within code that executes at an IRQ level greater than or equal to IRQL DISPATCH_LEVEL will result in a guaranteed system crash, despite the presence of exception handlers.

Finally, SEH should become pervasive throughout the driver implementation in order to have substantial benefits from its usage. If exception handling is not implemented systematically throughout the driver code base, the implementation will still be vulnerable to unexpected error conditions in the portions of the code that are not protected by exception handlers.

Structured exception handling is a methodology by which a developer or designer can provide exception handlers that can process exception conditions, avoiding the default processing performed by `KiDispatchException()`, namely, the call to `KeBugCheck()`.

NOTE No default handler exists to catch all kernel-mode exception conditions. Therefore, if your file system or filter driver causes an exception to occur but does not provide any exception handler to process such exceptions, you will bring down the system.

Furthermore, structured exception handling allows the designer to provide a systematic method for unwinding from within a specific block of code. This systematic unwinding can help ensure that exception conditions do not result in deadlocks or hangs or other similar nasty conditions because the thread cannot perform adequate cleanup processing when trying to recover from an unexpected error condition.

SEH requires compiler support; on Windows NT, a compiler that provides support for SEH is the Microsoft C/C++ compiler.

As I mentioned earlier in this chapter, an exception condition results in control being transferred to the kernel trap handler. The trap handler might resolve certain obvious exception conditions or it may in turn transfer control to `KiDispatchException()`, the exception dispatcher code within the NT kernel. `KiDispatchException()` in turn invokes `RtlDispatchExceptionO` to invoke any exception handlers that the developer might have provided. Your exception handler must be registered with the run-time library for it to be invoked. This registration is performed transparently by the Microsoft C/C^{++} compiler, which generates appropriate code to achieve this whenever it encounters the `try-except` construct (described below) in your code. Similarly, automatic unwinding of your stack-based call frame is performed whenever an exception condition occurs and you have used the `try-finally` construct in your code. The C/C++ compiler, cooperating with the run-time library, generates appropriate code for unwinding the call frame.

Here are the two primary constructs of structured exception handling:

- The `try-except` construct allows you to create code that can handle unexpected events or exception conditions cleanly, and is defined as follows:

  ```
  try {
      ...
      // Execute any code here.
      ...
  } except ( /* call an exception filter here. */ ) {
      ...
      // Code executed only if the exception filter returns
      //a code of EXCEPTION_EXECUTE_HANDLER.
      // This code is called the exception handler code.
      // Once this code is executed, control is transferred to
      // the next instruction following the try-except construct.
      ...
  }
  ```

  This construct consists of three parts: the try construct that allows you to define a block of code that is protected by your exception handler; the *exception filter* allows you to specify whether you wish to handle a specific exception condition; and the *exception handler* performs any exception condition-related processing.

- The `try-finally` construct allows you to specify a termination handler for a specific block of code. By doing so, you can ensure that correct cleanup-related processing is always performed, regardless of the method chosen to exit from the specific block of code. This construct is defined as follows:

```
try {
    ...
    // Execute any code here.
    ...
} finally {
    // Perform any cleanup here. The code within the finally
    // construct (also called the termination handler) will be
    // executed irrespective of the method chosen to exit from
    // the block of code protected by the try construct above.
}
```

  The `try-finally` construct consists of two parts: the try construct that allows you to define a block of code that is protected by the termination handler, and the `finally` construct, which contains the code comprising the termination handler itself.

The try-except Construct

The `try-except` construct allows you to protect a block of code such that, if an exception occurs within the protected block of code, control is transferred (by the RtlDispatchException() routine) to your exception handler. In order for this transfer of control to take place, the compiler and the Windows NT Kernel have to cooperate.

The compiler, upon encountering a `try-except` construct, automatically inserts additional code that registers an exception handler for that particular block of code (called a frame) with the NT Kernel. As described earlier in this chapter, the kernel is then responsible for assuming control when an exception condition occurs, and subsequently, the kernel allows your exception handler to take a crack at handling the exception.

Every exception that occurs as a direct result of executing code within the frame protected by the exception handler results in an eventual transfer of control to the exception handler code, unless the exception is handled by the attached debugger. However, you may not want your exception handler to handle all possible exception conditions; therefore, you can utilize an *exception filter* to determine whether your code should handle the exception or not.

The exception filter is the portion of code that is bracketed following the `except` keyword. Note that the exception filter can be fairly complex and you can actually invoke a function called the *exception filter function* to perform ai processing that is part of your exception filter. The exception filter can return 01 of three values:

- EXCEPTION_EXECUTE_HANDLER
- EXCEPTION_CONTINUE_SEARCH
- EXCEPTION_CONTINUE_EXECUTION

The EXCEPTION_EXECUTE_HANDLER return code value causes the excepti< handler to be executed. After the exception handler has completed its processir execution flow resumes at the first instruction immediately following the exce tion handler. To understand this better, consider these sample routines:

```
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 = 'B';

    return(0);
}
```

As you can see in the code fragment, an exception condition will occur when MyProcedure_B attempts to copy a character using the input argument character pointer. Since MyProcedure_B does not have an exception handler, the handler in the calling function (MyProcedure_A) will be invoked. The exception filter in MyProcedure_A is trivial, it immediately returns EXCEPTION_EXECUTE_HANDLER. This results in the exception handler being invoked, which simply obtains the exception code (STATUS_ACCESS_VTOLATION) and issues a debug print call.

The interesting point to note is that execution flow (after the exception handler code has executed) resumes at the statement AnotherlnaneVariable = 10 in MyProcedure_A. All statements following the one that caused the exception in MyProcedure_B are skipped and so are any statements that follow the invocation of MyProcedure_B within the function MyProcedure_A. This resumption of execution at the instruction immediately after the exception handler code is achieved by unwinding of the stack-based call frames.

Although the exception filter was extremely trivial in the preceding example, it can potentially be quite complex. You can invoke a separate filter function to determine what you wish to do with the exception condition, but remember that the filter function must return one of the three status codes listed above. As an argument to the filter function, you can pass the exception code using the GetExceptionCode ( ) intrinsic function call or you can use the GetExceptionInformation() intrinsic function call to pass even more information, such as the thread context represented by the contents of the processors' registers.

The GetExceptionCode ( ) intrinsic function returns the exception code value;* this function can be invoked either within the exception filter or within the exception handler, while the GetExceptionInformation() intrinsic function can only be invoked within the exception filter.

Typically, your exception filter (or any filter function that you use) will not require the additional information contained in the EXCEPTION_POINTERS structure, returned by the GetExceptionInformation() function. Although it is theoretically possible to modify the contents of individual registers contained within this structure, doing so results in extremely nonportable, and probably nonmaintainable, code; I would highly discourage it.

Note that neither the GetExceptionCode () function call nor the GetExceptionlnformation () call can be invoked from the exception filter function.

---

* It actually returns the same value that you could obtain from the ExceptionCode field in the EXCEPTION_RECORD structure defined later.

The following code fragment demonstrates the use of the exception filter function:

```
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 (GetExceptionCodeO ,
                    GetExceptionlnformation ( ) ) ) {
        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);
}
```

Exception handlers can be nested, either across procedure calls or even within the same function. Note that certain exception conditions are fatal errors (e.g., accessing paged memory that causes a page fault at an IRQL greater than or equal to DISPATCH_LEVEL) and will result in a system panic,* regardless of the fact that you had inserted exception handlers in your code. Therefore, as mentioned earlier in this chapter, do not assume that using exception handlers will guarantee that all error conditions can be effectively trapped.

The try-finally Construct

The `try-finally` construct represents a *termination handler* and is used to ensure consistent unwinding from within a block of code, even when exception conditions cause abrupt transfer of control to some other call frame. The concept here is very simple: consider a block of code protected within a `try-finally` construct. Before control is transferred to any instruction outside of the `try-finally` construct, statements enclosed within the `finally` portion of the construct are executed.

This simple example illustrates the point:

```
NTSTATUS MyProcedure_A (
int               *SomeVariable,
int               AnotherVariable)
{
    char             *APtrThatWasNotInitialized = NULL;
    int                AnotherInaneVariable = 0;
    int                AnotherInaneVariable2 = 0;
    NTSTATUS         RC = STATUS_SUCCESS;
```

---

* The VMM will bugcheck the system when it notices that the page fault was incurred at high IRQL.

```
    try {

        if (!AnotherVariable) {
            AnotherlnaneVariable = 7;
        }

        *SomeVariable = MyProcedure_B(APtrThatWasNotlnitialized,
                                              &AnotherInaneVariable)   ;

        // The following line is not executed if an exception occurs
        //in the procedure call above.
        AnotherInaneVariable2 = 5;

    } except (MyExceptionFilter(GetExceptionCode ( ),
        GetExceptionInformation())) {
        // Even if an exception condition got us here, the value of
        // AnotherlnaneVariable MUST be 15 since the statements within
        // the finally portion in MyProcedure_B get executed before we
        // start executing any code within the exception handler.
        ASSERT(AnotherlnaneVariable == 15);
        RC = GetExceptionCode ( ) ;
        DbgPrint( "Exception with value = 0x%x\n", RC) ;
    }

    //I will assert that AnotherlnaneVariable is set to 15 because the
    // assignment is within the "finally" construct in MyProcedure_B and hence
    // the assignment operation MUST have been performed.
    ASSERT(AnotherlnaneVariable == 15);

    // Execution flow resumes here once the exception has been handled.
    AnotherlnaneVariable = 10;

    return(RC) ;
}

int MyProcedure_B (
char              *IHopeThisPtrWasInitialized,
int                 *AnotherInaneVariable)
{
    char        ACharThatlWillTryToReturn =  'A';

    try {

        if (*AnotherInaneVariable ==7 ) {
            // This is a BAD thing to do if you value system performance.
            // However, I am executing a return here to illustrate that the
            // code within the "finally" below will get executed even though
            // I am abruptly returning from this function call.
            return(1);
        }

        // Exception occurs here if IHopeThisPtrWasInitialized is invalid.
        *IHopeThisPtrWasInitialized = ACharThatlWillTryToReturn;
        II The following code is never executed if an exception occurs
        // above.
        ACharThatlWillTryToReturn = 'B';

    } finally {
        // Whatever happens above, i.e., whether a return statement was executed
        //or whether an exception condition occurred, the following code will
        // get executed. Note that if an exception did occur, the following
        // code will get executed BEFORE the code within the exception
        // handler in MyProcedure_A gets executed.
        *AnotherInaneVariable = 15;
    }

    return(0);
}
```

Code for the exception filter function MyExceptionFilter() was presented earlier while discussing the try-except construct.

There are three different ways that flow of control can be transferred out of MyProcedure_B:

- It is possible that the return statement within the try-finally construct does not get executed and no exception condition occurs.

  This would happen if, for example, AnotherVariable is initialized to a valid value.

  In this case, all of the statements between the try and the finally keywords will get executed; then the code within the finally construct that comprises the termination handler will execute and then return control to MyProcedure_A via the return(0) statement.

- Consider the case where AnotherVariable is set to 0. Now, the return(1) statement within MyProcedure_B will return control back to MyProcedure_A. However, due to the presence of the try-finally construct, the code within the finally portion will get executed before the return(1) statement is processed. Therefore, the value of *AnotherInaneVariable will be set to 15.

- Now, consider the case where AnotherVariable is not set to 0, and APtrThatWasNotlnitialized is set to NULL. We know that this will result in an exception condition (STATUS_ACCESS_VIOLATION) in MyProcedure_B. You are also aware that MyProcedure_A has an exception handler that will process this exception since the exception filter used by MyProcedure_A is willing to handle exceptions of this type.

  However, before the exception handler gets executed in MyProcedure_A, the kernel unwinds the stack-based call frames. Part of this unwinding involves execution of any statements within the termination handlers* that protect any of the frames comprising the calling hierarchy between MyProcedure_A and MyProcedure_B.

In our example, MyProcedure_A directly invokes MyProcedure_B and so there is only one frame to unwind from and one corresponding termination handler.

Since the affected block of code (in which the exception occurred) is protected by a termination handler, the statements within the `finally` portion will be executed before statements comprising the exception handler in MyProcedure_A are executed. Therefore, the value of *AnotherInaneVariable will be set to 15 and we have an ASSERT in the exception handler in MyProcedure_A to check for this fact.

Typically, termination handlers are not used to perform the kind of simple initialization shown in the example. Rather, termination handlers are used to ensure that any necessary cleanup is performed before transferring control to some other module. For example, if memory had been allocated for some temporary purpose, freeing of this memory can be done within the termination handler (some BOOLEAN variable can be checked to see if memory had indeed been allocated or the value of the pointer itself can be checked if the pointer is always guaranteed to have been initialized to NULL).

Similarly, if any locks had been acquired (e.g., some ERESOURCE type of resource or some MUTEX had been acquired), the lock can be released from within the termination handler. Therefore, the termination handler is a powerful tool that can ensure that all required cleanup is performed in a consistent fashion and is guaranteed to be done, regardless of the method used to exit from the protected module.

A word of caution

In the preceding example, I placed a `return`(1) statement in the middle of MyProcedure_B in a block of code protected by a termination handler. The purpose of using this `return` statement was to demonstrate that even if such statements are used to exit the protected frame, the termination handler will still be automatically executed due to the stack-based call frame unwinding that takes place. This concept applies to other C statements, such as break and continue, which cause a transfer of control to some other statement. If this transfer of

---

* It is possible to prevent call frame unwinding in certain cases by inserting a `return` statement within the termination handler. However, this could result in completely indecipherable and unmaintainable code, and I strongly urge you not to even consider such esoteric usage and implementation of termination handlers.

control is to an instruction outside of the protected frame, the termination handler will always get executed before such a transfer of control takes place.

However, if you care about the performance exhibited by your driver, you should try never to use such statements in any frame that is protected by a termination handler. The reason for this is simple: call frame unwinding is an expensive operation in terms of execution time. In fact, on some processor architectures, unwinding can result in the execution of literally hundreds of extra assembly instructions. Therefore, you should try to avoid such unwinding, unless it happens because of some exception condition.

NOTE Exception conditions are, by definition, atypical events. Therefore, the prime consideration in dealing with an exception condition is to recover as gracefully as possible; performance considerations are secondary.

You might be concerned that avoiding the `return` statement in code that is protected by a termination handler could be very limiting. I would agree with that analysis; therefore, I will recommend that you use the following method to work around this limitation:

```
// Who says gotos are always bad ????
// Define the following macro in some global header file.
#define       try_return(S)       { (S) ; goto try_exit; }

NTSTATUS AnotherProcedureThatUsesATerminationHandler (void)
{
    NTSTATUS         RC = STATUS_SUCCESS ;
    ...

    try {
        ...
        if (!NT_SUCCESS(RC)) {
            // Assume for example that some memory allocation failed above
            // and that normal execution cannot continue.
            // Use the try_return MACRO here instead of a simple return
            // statement.
            // Note that any legitimate C statement can be executed as part of
            // the macro itself.
            try_return(RC);
        }
        ...

    try_exit:    NOTHING;

    } finally {
        ...
    }
    return(RC);
}
```

The `try_return` macro simply performs a jump to the end of the function (actually, it should cause a jump to the block of code just before the termination handler as illustrated in the code fragment). Additionally, it allows you to execute any legitimate C statement before the jump is performed. By using `try_return` instead of directly using a `return` statement, you can avoid the overhead of having the compiler perform call frame unwinding (to ensure that the termination handler code is executed) on your behalf.

By consistently utilizing the `try_return` macro in your code and also using termination handlers systematically, you can take complete advantage of the powerful functionality that termination handlers provide (especially with regard to consistent clean-up after error conditions that cause a premature exit from the frame) and yet not suffer from the performance degradation associated with call-frame unwinding.

Using both exception handlers and termination handlers together

Using both types of handlers together is the right way to both protect your code from unexpected exception conditions and also to ensure consistent clean-up within the frame. Typically, the following method can be employed:

```
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_exit:   NOTHING;
        } finally {
            // Clean-up code goes here.
        }
    } except (/* the exception filter goes here */) {
        //My exception handler goes here.
    }
    return(RC);
}
```

Event Logging

Often, kernel-mode drivers need to convey information to the system administrator or to the user of the host machine. This information could include error messages possibly caused by software or malfunctioning of attached hardware peripherals, warning messages that might indicate recovered errors or the possibility of an impending error, and informational (or status) messages indicating that some important activity had transpired.

The Windows NT Event Log serves as a central repository for messages sent by various software modules on that machine. The event log is a database containing event records that have a fixed, defined format. A user can either use the system-supplied event log viewer to extract information from the event log database or use the API supplied by the Win32 subsystem environment to obtain such data.

NOTE Although it might be possible to decipher the actual record structure of an event log entry in the file, if you wish to develop your own event log viewer, you might be better off simply using the Win32-based supplied API to access the event log file. This API includes calls to open the file, obtain individual records, and close the file.

The concept underlying the usage of the event logging facility in Windows NT is fairly simple. The kernel-mode driver logs an event indicating that something significant has occurred. Each event, which must be defined in a message file, has a unique event identifier associated with it. The event identifier has the same format as other NTSTATUS type of status codes (the format is described later). The logged event contains information such as the event identifier, the name of the component logging the event, or any strings or other binary data that should be associated with the event. The event log also allows the kernel module to include other pertinent information for events indicating an error condition, such as the number of times the operation has been retried (prior to logging the event), an offset in the device where the error occurred, the status that was returned by the driver to the I/O Manager in the I/O Request Packet (IRP), and other similar information.

Event identifiers have replacement strings associated with them. For example, the system-defined error IO_ERR_PARITY has the following replacement string associated with it (this string is typically read from the file *%SystemRoot%\system32\iologmsg.dll)'*

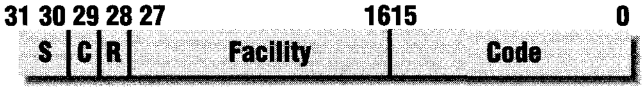
```
A parity error was detected on %1.t
```

---

* Note that *%SystemRoot%* is replaced by the location of the Windows NT installation of the host machine.

t This message also demonstrates the use of *insertion strings.* An insertion string is a string supplied by the driver when it records an event record. The first insertion string is represented as *%1,* the second as 962, and so on. The driver supplied insertion strings are automatically inserted into the text message by the event log viewer (which obtains the insertion strings from the event record).

The format of status codes returned by the system and of event identifiers created by independent drivers follows the figure below.



- S = Severity Code (2 bits). This can assume the following values:
  - — 00 = Success
  - — 01 = Informational
  - — 10 = Warning
  - — 11 = Error
- C = Customer Code Flag (1 bit). This bit should be set to 1 for status codes/ event identifiers defined by third-party drivers (which means that it should be set to 1 for any privately defined status codes in your driver).
- R = Reserved Bit (1 bit). Microsoft recommends that this bit be set to 0.
- Facility (12 bits). This indicates a group to which the error/status code belongs and can have one of the following values:
  - — FACILITY_NONE (defined as 0x0)
  - — FACILITY_RPC_RUNTIME (defined as 0x2)
  - — FACILITY_RPC_STUBS (defined as 0x3)
  - — FACILITY_IO_ERROR_CODE (defined as 0x4)

  If you develop file system or filter drivers or other such kernel-mode drivers, you will typically set the value of Facility to 0x0 or to 0x4 for any status codes defined by you.
- The Code (16 bits). This can be set to any unique value for each status code defined in your driver. A typical error code your driver might define follows:

  ```
  –define     MYDRIVER_ERROR_CODE_IO_FAILED          (OxE0047801)
  ```

  If we expand the error code, we get the binary value 1110 0000 0000 0100 0111 1000 0000 0001.

  This corresponds to the fact that this is an error condition (bit positions 30 and 31 are set to 1), this is a customer defined error (bit position 29 is set to 1), the reserved bit is set to 0, the facility that this error belongs to is FACILITY_IO_ERROR_CODE (since *Facility* is set to 0x4—bit positions 16 through 27), and the privately defined error code is 0x7801 (bit positions 0 through 15). Note that the privately defined error code can be any 16-bit value, as long as all such error codes defined by your driver are unique within your driver.*

So that the event viewer application (or any application that interprets the error log) can associate a textual description with your event identifier, your driver will have to supply a message file that contains the text associated with each particular event ID. For example, a typical text message that might be identified with our error defined previously could be as follows:

```
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).
```

As you can see, providing the user with a clear explanation of the probable cause of failure is quite helpful. The system administrator might be able to use the error message supplied by you as a starting point to diagnose and rectify the cause for the failure. An appropriate message file can be created using the resource compiler used by your driver. Consult the documentation supplied with your resource compiler to determine how to create the message file.t Here is an example message file:

```
;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=OxO:STATUS_SEVERITY_SUCCESS
               Informational=0xl:STATUS_SEVERITY_INFORMATIONAL
               Warning=0x2:STATUS_SEVERITY_WARNING
               Error=0x3:STATUS_SEVERITY_ERROR)
```

---

* Although the error code defined by your driver might also be (coincidentally) defined by some other driver in the system, event identifiers are uniquely identified as a tuple consisting of (source, event ID). Therefore, as long as all identifiers within your driver are unique, you should not be concerned about providing unique values with respect to all other drivers in the system.

t Note that your resource compiler, while processing your input file, can create both the output message file as well as a header file that you can use with your driver. Therefore, you should have to define the event identifiers (and the associated text) in only one place, the input file to your resource compiler. This will help prevent discrepancies between any header files that your driver might use and the message file that eventually ships with your driver.

```
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_
```

It is possible to have insertion strings within text messages associated with event identifiers. Placeholders are denoted as %1, %2, etc. If you specify placeholders in your message, the driver can supply the strings to be inserted when writing the event log entry. However, note that the I/O Manager always inserts the device/driver name as the first insertion string with every recorded event record. Even if the driver supplies insertion strings, they are placed after the device/driver name. The net result is that using %1 as a placeholder for an insertion string in your text (associated with an event identifier) will always result in the device/driver name being placed there, instead of the first driver-supplied insertion string. To obtain the first driver-supplied insertion string, use %2 as the placeholder; to get the second driver supplied insertion string, use %3, and so on.

How the Event Log Viewer Finds a Message File

For the event log viewer to be able to find your message file, your driver (or more likely, the application that installs your driver) will have to modify the Registry on the target machine. Typically, users will use the Win32 subsystem as their native subsystem. In this case, a unique subkey should be created by the installation utility under the Registry path: *CurrentControlSet\Services\EventLog\System.*'

---

* Note that there are three possible locations under the EventLog key where your subkey could potentially be located: *Application* (for applications or user-mode drivers), *Security,* and *System* (for system-supplied and kernel-mode drivers). As kernel-mode driver developers, the logical and correct choice for your entry is under the *System* key.

This unique subkey should have the same name as the driver executable. For example, the subkey for the system-supplied AT disk driver is *atdisk,* which is the same as the name of the driver executable file *(atdisk.sys).* Within this subkey, at least two value entries must be created:

`EventMessageFile`

> This value is of type *REG_EXPAND_SZ* and contains the complete path and filename for the message file that contains the text messages corresponding to each event identifier. An example of a complete pathname and filename might be *%SystemRoot%\MyDriverDirectory\message.dll.*

`TypesSupported`

> This value is of type *REG_DWORD* and should be set to 0x7 for your driver, indicating that your driver supports events of type *EVENTLOG_ERROR_TYPE* (defined as 0x1), *EVENTLOG_WARNING_TYPE* (defined as 0x2), and *EVENTLOG_INFORMATION_TYPE* (defined as 0x4).

Once you have created the appropriate Registry entries and your installation program has copied the appropriate message file (in our example: *message.dll~)* to the correct directory, the event log viewer application should be able to find and use the contents of your message file.

Recording Event Log Entries

Now that you have defined the appropriate event identifiers specific to your driver, you can use these event identifiers to record event log entries using support routines provided by the I/O Manager. Logging an event is performed in two steps:

1. An event/error log entry is allocated with `IoAllocateErrorLogEntry ()`.
2. After the error log entry has been initialized, the event can be logged with `IoWriteErrorLogEntry()`.

The routine used to allocate an event log entry, so it can subsequently be recorded, is defined as follows:

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

Parameters:

IoObject

This must point either to the device object* representing the device for which the error/event is being logged or to a driver object representing the driver controlling a device for which the event is being logged.

EntrySize

The size of the object to be allocated. Since you as the developer can log binary data with the event log record, and you can also supply insertion strings that augment your message, the size of the entry should be calculated as follows,

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

where n = number of words of data to be dumped with the event record.

Functionality Provided:

The `IoAllocateErrorLogEntry()` will allocate an entry for your use. You can then initialize this entry and invoke `IoWriteErrorLogEntry()` to write to the event log. This routine returns a NULL pointer if it cannot allocate an entry for your use. In this case, you should not write the event at this time and wait for the next occurrence of the error, then you can retry this operation.

One note of caution: this routine references the device/driver object passed in as the IoObject argument. Therefore, once you invoke this routine, you must invoke the `IoWriteErrorLogEntry()` routine, which will dereference the object and release the memory allocated for the error log entry.

Initialization of the error log entry is quite simple and is well documented in the device driver reference supplied by Microsoft.

Once an event log entry has been obtained, you must invoke the IoWriteErrorLogEntry() routine to write the record to the event log. This routine is defined as follows:

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

Parameters:

ElEntry

The initialized event log entry to be recorded.

---

* See Chapter 4, *The NT I/O Manager,* for a discussion on device object and driver object structures.

Functionality Provided:

The loWriteErrorLogEntry ( ) queues the initialized event log entry to be written out. The actual write operation will be performed asynchronously by a system worker thread. Note that since this routine returns immediately after queuing the entry, the device/driver object will not yet have been dereferenced. The dereference will only occur after the entry has been asynchronously written to the event log.

NOTE The -way that the entry is asynchronously written to the event log file is also interesting. The system worker thread dequeues the event log record, inserts the device/driver name strings and then uses LPC (a local procedure call) to write the record to a special port where another user space thread writes the entry to the event log file. The system worker thread continues writing out all records until it encounters an error condition or until all the pending event log records have been sent to the port handler.

Driver Synchronization Mechanisms

One of the primary functions of a driver is to prevent data corruption. A principal cause of data corruption is a lack of synchronization between two or more concurrent threads of execution that manipulate the same data structures. Since Windows NT can execute on both uniprocessor as well as on symmetric multiprocessor machines, it becomes especially important for kernel-mode drivers to use synchronization primitives carefully. If you develop device drivers, take care when manipulating data structures shared by Interrupt Service Routines (ISRs) and threads that execute at normal IRQ level.

As discussed in Chapter 1, *Windows NT System Components,* the Windows NT kernel-mode environment contains the Executive as well as the Windows NT Kernel. The Executive is preemptible and parts of it are also pageable. The Kernel, however, is neither preemptible nor pageable. Although not preemptible, on multiprocessor machines, the kernel can execute concurrently on each processor.

In this section, we'll see the various synchronization primitives available to drivers forming part of the NT Executive. These synchronization primitives are either exported by the NT Kernel itself or by the NT Executive; the Executive uses the basic primitives supplied by the Kernel to construct more complex synchronization primitives. My intent is to provide an introduction to the different primitives available and to explain where each can be used. The sample code in the remainder of this book, especially file system and filter driver code, will make extensive use of some of these synchronization primitives. For further information on the syntax for calling the supporting routines mentioned in this section, consult the Microsoft Device Drivers Kit (DDK) documentation.

Spin Locks

The kernel spin lock structure is fundamental to providing synchronization across processors in a multiprocessor environment. Spin locks are used to provide mutual exclusion, i.e., a spin lock is used to ensure that only one thread executing on one processor can access the shared data protected by the spin lock. The sequence of instructions executed after acquiring the spin lock is also known as a *critical region.* The critical region ends when the spin lock is released.

When a thread on a processor acquires a spin lock, context switching (preemption of the thread) is disabled until the thread releases the spin lock. Similarly, any other thread, executing on another processor, will continuously try to gain access to the spin lock, making no progress until it succeeds. This method of busy-waiting (i.e., continuously attempting to check to see if the spin lock has become available) is also called *spinning* for the lock, hence leading to the name spin lock.*

The exact method used by the kernel to implement a spin lock is processor dependent; typically, however, an atomic *test-and-set* assembly instruction is used to implement the spin lock. The software tests the state of the lock variable and if it's free, sets it to the busy state. If the lock state is busy, the software keeps repeating execution of the test-and-set instruction.

NOTE Often, to reduce bus contention, the test-and-set operation is not used continuously in the implementation of the spin lock. Rather, the operating system uses the test-and-set instruction once, and if it finds the lock state set to busy, then ordinary polling instructions are used until the lock state is found to be free. At this time, the test-and-set instruction is retried to obtain the lock.

Spin locks must be acquired by the thread executing on a processor at the highest IRQL at which all other attempts to acquire the same spin lock will be made.

Follow these few, simple rules to ensure correct behavior of your spin locks:

---

* The thread that is spinning for the lock cannot be preempted (just as the thread that has acquired the lock). However, these threads can be interrupted by an interrupt at a higher IRQL than the one at which they execute.

- Never refer to any pageable data once your code acquires a spin lock. Similarly, all code that executes once a spin lock is acquired must be nonpageable code.

  The reason for this restriction is that the system cannot service any page faults at an IRQL greater than or equal to the DISPATCH_LEVEL. Therefore, when a page fault occurs at a high IRQL, the system checks for pageable code and issues a KeBugCheck ( ) , assuming that the condition is a direct result of a programming/design error.

- Try not to call other functions once you have acquired a spin lock. If you do need to call another function, be sure that none of the functions called refer to any pageable code or data.

- Because spin locks must be shared by all processors on a node, keep the spin lock for the minimum amount of time possible, then release it to allow another processor to acquire it.

It is possible to design and implement code in which a sequence of spin locks are acquired, i.e., you acquire spin lock #\ followed by a call to acquire spin lock *#2,* and so on. Or else, you might implement code in which you acquire a spin lock and then follow this up with one or more calls to acquire other synchronization primitives (e.g., mutexes). This could cause a deadlock condition. There is no deadlock checking performed when a processor acquires multiple spin locks and since dispatching or preemption is disabled once any spin lock is acquired, it is quite possible to create a system deadlock.

There are two types of spin locks that exist on Windows NT platforms:

Interrupt spin locks

These spin locks synchronize access to device driver data structures. They are acquired and released at the IRQL associated with the particular device managed by the device driver. The device driver usually does not allocate memory for spin locks itself, neither does it explicitly acquire or release interrupt spin lock structures. As a matter of fact, the kernel automatically acquires the spin lock associated with the interrupt before invoking the Interrupt Service Routine (ISR) for the interrupt, and releases it after the ISR execution has been completed.

The KeSynchronizeExecution( ) function, documented in the DDK, can be used to synchronize the execution of a device driver routine with the execution of an ISR for a specific interrupt. This function acquires the interrupt spin lock associated with the interrupt pointer, supplied as an argument to the routine after raising the IRQL for the thread to the DIRQL for the interrupt. KeSynchronizeExecution( ) then invokes the specified routine whose execution has to be synchronized with that of the ISR and finally releases the spin lock before returning to the caller.

Interrupt spin locks must always be used whenever a lower level driver wishes to synchronize the execution of a module with the ISR for the driver. Attempting to use an Executive spin lock will undoubtedly lead to data corruption and/or system deadlock situations.

Executive spin locks

Executive spin locks can only be acquired from threads executing at IRQL PASSIVE_LEVEL, APC_LEVEL, or DISPATCH_LEVEL. Therefore, they are typically used by higher level drivers such as file system drivers to synchronize access in multiprocessor environments. They can certainly be used by device driver developers, as long as they are not used to synchronize execution with the ISR for the drive driver.

The rest of this discussion assumes that you are using Executive spin locks to synchronize data access.

For the remainder of this book, we will focus on the usage of Executive spin locks.

To use Executive spin locks, you must first allocate enough storage for a spin lock structure. The storage for a spin lock must be allocated from nonpaged pool. Typically, your driver should either embed the spin lock definition in the driver extension, which is always allocated from nonpaged memory, or use a global definition, since all global variables within a kernel-mode driver are typically not pageable; or use an allocation function (e.g., `ExAllocatePool (NonPagedPool, sizeof (struct KSPIN_LOCK) ))`.

WARNING If you happen to allocate a dispatcher object from paged pool instead of nonpaged pool, you will see some unexpected system bugchecks occur. Your driver might be working fine, but occasionally the system will bugcheck with an exception indicating that paged memory was accessed at high IRQL. The stack trace that you might obtain will not even point to your driver. This is because the kernel stores all threads waiting for active dispatcher objects on global linked lists. Each of these linked lists is protected by a spin lock. When the kernel traverses such a linked list with the spin lock acquired, and the object on the list happens to be paged out, you will encounter a system bugcheck. Note that the object might not always be paged out of memory, so your system might work fine, sometimes, though not always.

The following kernel support routines are available to you for manipulating an Executive spin lock:

KeInitializeSpinlock()

This routine accepts a pointer to the allocated spin lock structure. It will initialize the spin lock, and must be invoked before trying to acquire the spin lock for the first time.

`KeAcguireSpinLock() /KeAcguireSpinLockAtDpcLevel()`

These routines will spin trying to acquire the spin lock. A pointer to the spin lock to be acquired must be passed in as an argument. When either of these routines returns, the spin lock will have been acquired. The only difference between the two routines is that the KeAccruireSpinLock() routine will first raise the IRQL for the processor to DISPATCH_LEVEL and therefore return the old IRQL to the caller, to be used in releasing the spin lock, while KeAcquireSpinLockAtDpcLevel () assumes that the current IRQL is already at DISPATCH_LEVEL.

NOTE On uniprocessor systems, the `KeAcquireSpinLockAtDpcLevel ()` doesn't do anything, i.e., it immediately returns control to the caller. Therefore, invoking this function (if appropriate) will result in a slight performance gain for your driver on uniprocessor systems.

`KeReleaseSpinLock() /KeReleaseSpinLockFromDpcLevel{)`

These routines allow you to release a previously acquired spin lock. KeReleaseSpinLock () expects an additional parameter: the old IRQL returned from the previous call to `KeAcquireSpinLock()`. The processor is returned to the old IRQL, once the spin lock is released.

A spin lock should be used whenever synchronization is required across multiple processors, in arbitrary thread contexts, when processing interrupts, and when context switching has to be prevented. Furthermore, all the rules mentioned earlier should be followed whenever spin locks are used. If, however, you wish to provide synchronization across multiple processors in the context of some thread and you do not mind context switching occurring, then other Executive dispatcher objects (described in the next section) can be used.

Note the words *in arbitrary thread contexts* in the preceding paragraph. Spin locks can be used even by device drivers, whose entry points (such as read/write) are typically executed in the context of some arbitrary thread. It is even probable that such entry points for device drivers might be executed at high IRQL. Spin locks can be used freely by such drivers, while other dispatcher objects (such as mutexes or event objects) can only be used in a nonarbitrary thread context, i.e., the other dispatcher objects are used by file system drivers or filter drivers that sit above the file system.

NOTE File system driver dispatch routines (e.g., read/write routines) are typically executed in the context of a system worker thread for asynchronous operations or in the context of the user thread initiated by an I/O request (e.g., a user application invoked the ReadFileO call). Since this is a nonarbitrary thread context, file systems are free to wait for dispatcher objects to be set to the signaled state.

Device drivers, on the other hand, have IRPs (I/O Request Packets) queued. Each I/O Request Packet for a driver dispatch routine is subsequently dequeued (and the request initiated), in the context of whichever thread happens to be currently executing on that processor. Therefore, since the dispatch routine for the device driver executes in the context of an arbitrary, unknown thread, •waiting for dispatcher objects to be signaled is not allowed. Thread context is discussed in detail later in this book.

Dispatcher Objects

Kernel dispatcher objects are a set of abstractions, provided by the kernel to the Executive, to support synchronization. These objects control dispatching and synchronization of system operations. Dispatcher objects can be in one of two states:

- *Signaled* state, in which no thread is currently accessing the shared data protected by the dispatcher objects or no other thread is currently within the critical section of the code.
- *Not-signaled* state, indicating that a thread is accessing shared data protected by the dispatcher objects and/or executing the critical region of the code.

Since your driver forms part of the NT Executive, you can use these dispatcher objects to provide synchronization within your driver implementation. Note that dispatcher objects provided by the kernel must be treated as opaque data structures. The kernel provides all functions that you might need to initialize, query the state, set the state, and clear the state for these objects. You must provide the storage needed to contain these objects. This storage must be provided from nonpaged pool (similar to that provided for spin locks) and can be provided from the driver extension structure, as a global variable, or as memory that is allocated dynamically.

The method used to synchronize access to shared data or to control execution within a critical region of code follows:

1. A thread needs to access a shared data resource (i.e., access some shared data or execute code within a critical region, so it invokes a Kernel Wait Routine).

   The wait routines that the thread can invoke are:

   – `KeWaitForSingleObject()`

   – `KeWaitForMultipleObjects()` or

   – `KeWaitForMutexObject()`

   If the objects being waited for are in the signaled state, the wait will be satisfied and control will return to the waiting thread. Note that before the wait is satisfied, the objects that were being waited for will be set to the not-signaled state, preventing any other thread, which might concurrently invoke a wait routine, from simultaneously getting access to the shared data resource.

2. Any other thread invoking a wait routine for one of the objects set to the not-signaled state in Step 1 will be suspended.

3. When the first thread completes processing the shared data resource, it will invoke an appropriate routine, depending upon the object used to achieve synchronization, KeReleaseMutexO or KeSetEvent(), to release the dispatcher objects and set the state of the dispatcher object to Signaled.

4. Now that the first thread has released the dispatcher objects, one of the threads waiting for the dispatcher objects, to gain access to the shared data resource, will be awakened.

   This thread will now be permitted access to the shared data resource.

NOTE In the case of some synchronization objects, multiple threads will be awakened concurrently. However, only one of them will subsequently be able to acquire the synchronization object.

---

5. Steps 1 to 4 are repeated every time a thread wishes to access the shared data resource.

If a thread cannot gain access to the shared data resource (i.e., if the dispatcher object is in the not-signaled state because another thread is actively accessing the shared data or executing code within the critical region), the thread will be suspended or blocked, awaiting the release of the dispatcher object. This allows other threads in the system to continue executing and is very different from a spin lock, where the thread will be in a busy-wait state until it gains access. Dispatcher objects are therefore more conducive to better system performance.

As mentioned earlier in the discussion on spin locks, driver dispatch routines that execute in an arbitrary thread context cannot wait for dispatcher objects to be signaled. Also, it is considered a fatal error to wait for a nonzero time interval on a dispatcher object at IRQL that is greater than PASSIVE_LEVEL. Therefore, most device driver designers will not use dispatcher objects for mutual exclusion, but file system developers or developers of filter drivers that sit above the file system in the calling hierarchy can potentially use dispatcher objects.

Finally, note that when a thread invokes the kernel routine to wait for a dispatcher object (e.g., `KeWaitForSingleObject()`), the thread can specify a TimeOut interval. If the dispatcher object does not get signaled within the specified TimeOut interval, the thread will be awakened with a special status code of STATUS_TIMEOUT. This allows the thread to ensure that the wait will be a bounded one. If a TimeOut interval of 0 is supplied, the thread will never be put to sleep; the state of the object will be checked and if not-signaled, control will immediately be returned to the thread with the status of STATUS_TIMEOUT.

The following dispatcher objects are available to designers and developers of kernel-mode drivers:

- Event objects
- Timer objects
- Mutual exclusion objects
- Semaphore objects

In addition to the dispatcher objects listed here, threads can also wait for process, thread, and file object structures.

Event objects

Event objects are used to synchronize execution between multiple threads. They record the occurrence of an event that determines execution flow. Consider a producer-consumer relationship between two threads: producer thread A creates data to be processed while consumer thread B processes data whenever it becomes available. Since thread B does not know when data will become available, it has two options:

- Keep inquiring from thread A whether data is available for processing. This is not conducive to good system performance, since valuable processor cycles get consumed in this kind of busy-wait mode.
- Wait for thread A to inform it whenever data is made available.

Since the second option is clearly superior, it is most often used in such situations.

To implement this notification, an event object can be used.*

---

* Note that a counting semaphore (discussed later) could he used equally well for this purpose.

The event object must be initialized before it can be used. Initially, the event object would be set to the not-signaled state. Thread B would then invoke a wait call on this event object and would be suspended from execution until the wait can be satisfied. When thread A has data available for processing, it could invoke the KeSetEvent() call to signal the event object. This would result in thread B being inserted into the queue of threads that can be scheduled for execution. At some point, the system scheduler schedules thread B for execution, and thread B processes the data. This method can be repeated as often as required.

There are two types of event objects:

Notification event objects

In this type of event object, every thread that is waiting for the event object is scheduled for execution once the event object is signaled. Also, the state of the event object, when signaled, is not automatically reset to the not-signaled state. Therefore, an explicit call to KeResetEvent() will have to be made by some thread to set the state of the event object to the not-signaled state.

This type of event object is typically used when a single occurrence of an event, resulting in the event object being set to the signaled state, triggers execution by any other thread waiting for that event to occur. For example, consider the analogy of a car race: when the start signal is given, all cars in the competition take off, each trying to get to the finish line.

Synchronization event objects

This type of event object is our producer-consumer example. Here, when the event object is set to the signaled state, only one waiting thread will be scheduled for execution and the event object is then automatically reset to the not-signaled state. This type of object ensures that only one thread accesses the shared data resource at any point in time.

The following kernel-mode support routines are available for interacting with event objects:

KeInitializeEvent()

Your driver must allocate storage for the event object from nonpaged pool. Once you have allocated the storage, you must invoke this routine to initialize the event object before any thread attempts to wait for, signal, or reset it. When this routine is invoked, you can specify whether the event object should be a notification type object or a synchronization type object. You can also specify the initial state of the event object, signaled or not-signaled.

IoCreateSynchronizationEvent()

Note that this is not strictly a kernel support routine, but one provided by the I/O Manager. This routine is only available in the Windows NT 4.0 and later releases and allows your driver to request that a named synchronization event object be created or opened. Since this event object has a name, multiple drivers can now use the same event object to synchronize access to a shared data resource.*

This routine will either create a named event object, if no such event object exists (and also initialize the event object to the signaled state), or open a previously created event object. It returns two values, a pointer to and a handle for the event object. All the calls to manipulate event objects, listed below, can be used on the returned event object pointer. When your driver no longer needs to use this object, it should invoke the `ZwClose()` routine to close the returned handle.

`KeSetEvent()`

This routine allows you to set the state of the event object to the signaled state. One or more threads that are waiting for the object to be signaled will get scheduled for execution.

Consider the following pseudocode fragment:

```
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;
    }
}
```

This code describes a typical producer-consumer relationship. Here we see that each thread performs a wait operation immediately after signaling an event object.

Signaling an event object is one point when the system scheduler might perform a context switch. However, since our threads will voluntarily put themselves to sleep following the signal operation, it seems redundant for them to be scheduled out, only to be rescheduled some time later and immediately put to sleep. It would be more efficient, instead, if they were allowed

---

* The event objects that you otherwise allocate storage for from within your driver are only accessible to your driver unless you implement some horrendous method of passing pointers between drivers, using a private IOCTL. Therefore, it was quite difficult for two or more drivers in Windows NT Version 3.51 and earlier versions to synchronize access to a shared data resource using event objects.

to continue executing after the signal operation so that they could put themselves to sleep and avoid the extra overhead of one unnecessary context switch. This can be achieved by specifying the Wait argument to KeSetEventO as TRUE.

NOTE Implementation of POSIX threads-style condition variables requires the capability to atomically release a mutex object and then put the thread that released the mutex to sleep. This can be achieved easily by specifying the Wait argument in KeSetEvent () as TRUE.

KeResetEvent()/KeClearEvent()

Both routines allows you to set the state of the object to the not-signaled state. KeResetEvent ( ) also returns the previous state of the event object.

KeReadStateEvent( )

This routine gives you the current value of the event object (signaled or not-signaled).

Timer objects

Timer objects are used to record the passage of time. If a thread wishes to perform a task after some time has elapsed or at a specified time value, it should use a timer object. The timer object has a state associated with it, either signaled or not-signaled. When the desired time interval passes, the timer object is set to the signaled state and all threads waiting for the timer object will have their wait satisfied and will be scheduled for execution.

Just as with other dispatcher objects, storage for the timer object must be provided in nonpaged memory by the driver. The timer object must be initialized to the not-signaled state.

There are two ways your driver can use a timer object:

- A thread in your driver might initialize a timer object and then invoke a wait routine (e.g., KeWaitForSingleObject ( )) to suspend execution until the timer object is set to the signaled state (after the specified time interval has elapsed).
- Alternatively, when setting the timer object to expire after the time period has elapsed, a Deferred Procedure Call (DPC) might be specified. When the time period expires, the DPC routine will be scheduled for execution, and any required processing could be performed within that DPC routine.

NOTE Deferred Procedure Calls are another way of influencing the operation of the kernel. The DPC provides the capability of breaking into the execution of the currently running thread (via a software interrupt), and executing a specified procedure at IRQL DISPATCH_LEVEL. No system services can be invoked when executing the DPC procedure and page faults are not tolerated. Furthermore, DPCs are not targeted to a specific thread like Asynchronous Procedure Calls. Whenever the current IRQL falls below DISPATCH_LEVEL, a software interrupt will happen and the DPC dispatcher invoked. Typically, DPCs are used by device drivers to complete interrupt-handling-related processing.

On any single processor, only one DPC can be executing at any given instant in time. However, on multiprocessor systems, there could potentially be a DPC executing on each processor concurrently. Thread scheduling on the processor is suspended while the DPC is executing at IRQL DISPATCH_LEVEL

With the release of Windows NT 4.0, two types of timer objects are available:*

Notification timer object
: When this type of timer object is signaled, all threads that were waiting for this object have their waits satisfied. These threads will all get scheduled for execution.

Synchronization timer object
: When this type of timer object is signaled, only one thread waiting for the timer object will have its wait satisfied. The timer object will automatically be reset to the not-signaled state.

One further enhancement made in Windows NT 4.0 to timer objects is that you can now specify periodic (recurring) timer objects. These timer objects will automatically be reinserted into the active timer list, as many times as specified in the `Period` argument when setting the timer object.

The following kernel-mode support routines are available for interacting with timer objects:

`KelnitializeTimer()/KeInitializeTimeEx()`

The latter version is only available on the Windows NT 4.0 and subsequent releases. This routine expects a pointer to a timer object allocated in nonpaged memory. It will initialize the value of the timer object to the not-signaled state. With the `KeInitializeTimeEx()` routine, you can specify the type of the timer object (Synchronization type or Notification type).

---

* Windows NT 3.51 and earlier versions only supported the notification type of timer object.

KeSetTimer()/KeSetTimerEx()

This routine allows you to set a timer object. The time value is specified in system time units (100-nanosecond intervals). You have two choices: if you supply a negative time unit value, the value will be interpreted relative to the current time when the routine was invoked. If a positive value is supplied, it is interpreted as an absolute value; the time that the system was booted is taken as time unit 0.

The KeSetTimerEx() routine became available with the release of Windows NT 4.0 and it allows you to specify the number of times you wish the timer to be reactivated.

Note that you can specify a DPC routine to be invoked once the timer is set to the signaled state.

KeReadStateTimer()

This routine returns the current state of the timer (signaled or not-signaled).

KeCancelTimer()

This routine cancels a previously set timer if it has not yet expired. If there is an associated DPC routine, it is canceled too.

Two points should be noted here. First, canceling a timer does not set the state of the timer to the signaled state. Second, if the timer had previously expired and the associated DPC routine is in the queue, that DPC routine will not get canceled. Only if the timer had not previously expired will the associated DPC routine not get queued.

Mutex objects (mutual exclusion)

Mutex objects are similar to spin locks in that they allow only one thread to access a shared data resource at any given instant in time. Any other thread that attempts to acquire the same mutex object will be suspended until the first thread releases the mutex object. The fact that a thread will be suspended awaiting the mutex object to be signaled is the distinguishing feature between spin locks and mutex objects.

Storage for mutex objects must be provided by the driver from nonpaged pool. Also, the driver must ensure that any code executed once a mutex is acquired is not pageable. Mutex objects come in two varieties:

Fast mutex objects

A fast mutex is simply a wrapped-up event dispatcher object. It provides mutual exclusion semantics by allowing only one thread to acquire the mutex at any instant. When the mutex object is released (i.e., its corresponding event is signaled), only one other thread from those waiting for the mutex object will be scheduled for execution. Therefore, the concepts underlying the fast mutex data structure are the same as those for synchronization type event object structures.

Fast mutex objects do not provide any form of deadlock prevention support. Also, fast mutex objects cannot be recursively acquired. Therefore, if you implement code in which one thread tries to acquire fast mutex #\ followed by fast mutex #2 while another thread does so in the reverse order, you will get a deadlock situation. Similarly, any thread that tries to recursively obtain a fast mutex will deadlock with itself.

Support for fast mutex objects is provided by the NT Executive, because fast mutex objects are not among the primitive synchronization mechanisms exported by the Windows NT Kernel. Using fast mutexes is faster (hence the name) than using the normal mutex structures supported by the kernel. The routines to manipulate fast mutex objects follow:

`ExInitializeFastMutex()`

Initializes the passed-in fast mutex structure. This is actually a macro that simply initializes the event object that comprises the fast mutex structure.

`ExAcquireFastMutex()/ExAcquireFastMutexUnsafe()`

If the fast mutex is not currently acquired by another thread, this thread will be allowed to acquire the fast mutex. Any other thread that subsequently attempts to acquire this mutex will be suspended until the mutex is released.

If the mutex had already been acquired by some other thread, the current thread will be blocked until the fast mutex becomes available.

The difference between the two invocations is simple: if ExAcquireFastMutex () is used, the Executive disables delivery of Asynchronous Procedure Calls (APCs) to the thread that has acquired the fast mutex. If ExAcquireFastMutexUnsafe () is used instead, the Executive assumes that the call is protected within a critical region* and hence does not bother to disable APCs.

---

* Highest level drivers such as file system drivers can invoke KeEnterCriticalRegion() and KeLeaveCriticalRegion () to note that the current thread is entering or leaving a critical region. Invoking `KeEnterCriticalRegion()` disables kernel-mode APCs. `KeLeaveCriticalRegion()` reenables delivery of kernel-mode APCs to the calling thread. The KeEnterCriticalRegion () macro should be invoked whenever your driver would find it awkward to be interrupted from its processing to receive a kernel-mode APC.

NOTE Asynchronous Procedure Calls are a method by which control flow for a thread can be affected. An APC must be targeted toward a specific thread. This is in contrast to a DPC, which executes in the context of any arbitrary thread currently executing on the processor.

The thread to which an APC is directed will be interrupted (via a software interrupt), and the procedure specified when creating the APC will be executed in the context of the interrupted thread at a special IRQL, APC_LEVEL.

APCs can be delivered both in user mode and in kernel mode. Kernel-mode APCs come in two flavors: *normal* and *special.* Normal APCs can be disabled by a kernel-mode driver by invoking KeEnterCriticalregionO. However, special APCs cannot be disabled. Consult the DDK for more information on Asynchronous Procedure Calls.

ExReleaseFastMutex()/ExReleaseFastMutexUnsafe()

These calls release a previously acquired fast mutex. Note that the appropriate call to be used depends on which call was invoked to acquire the fast mutex, `ExAcquireFastMutex()` or ExAccruireFastMutex-Unsafe().

ExTryToAcguireFastMutex()

This routine will attempt to acquire the fast mutex. If it is successful, it will return TRUE (and will have blocked kernel-mode APCs). If it could not acquire the fast mutex, it will return FALSE. The caller then has the option of retrying immediately (polling) or retrying after some period of time.

Mutex objects

Mutex objects are similar to their fast mutex counterparts. However, mutex objects are supported by the NT Kernel, and they have the following additional features missing in the fast mutex implementations:

- Your driver can associate a level with each mutex object that it initializes.[*]

  The kernel checks the level of the mutex being acquired to ensure that all previously acquired mutexes are at a level strictly less than the level of the current mutex (unless the same mutex is being acquired recursively).
- Mutex objects can be acquired recursively.

---

[*] The level associated with a mutex object should correspond to your locking hierarchy. For example, if your locking hierarchy dictates that mutex #1 is always acquired before mutex #2, then you should associate a lower level (lower nonzero numerical value) with mutex #1 and a higher level (higher nonzero numerical value) with mutex #2.

Therefore, a thread in your driver can safely reacquire the same mutex object multiple times. The only restriction is that the mutex should be released exactly the same number of times that it was acquired.

- When a thread in your driver has a wait on a mutex object satisfied, the priority of the thread is boosted to the lowest real-time priority in the system.

  This priority will subsequently automatically be lowered when the mutex object is released.

- The owning process (for the thread that acquires the mutex) will not be paged out to secondary storage.

The following routines are provided by the NT Kernel to support mutex objects:

`KeInitializeMutex()`
: Your driver must specify a valid nonzero `Level` argument if it needs to acquire multiple mutex objects concurrently (if you specify 0 as the value for `Level` for each mutex that you initialize, trying to acquire multiple mutex objects concurrently will result in a system bugcheck).

`KeReadStateMutex()`
: This routine returns the current state of the mutex (signaled or not signaled).

`KeReleaseMutex()`
: This routine is used to release a previously acquired mutex. If the thread releasing the mutex expects to immediately execute a call to a kernel wait routine (e.g., `KeWaitForSingleObject()`), it should supply the Wait argument as TRUE. This will avoid an unnecessary context switch.

Semaphore objects

Semaphore objects (counting semaphores) allow one or a specific number of threads to concurrently access a shared data resource. They can be used to provide mutual exclusion (similar to mutex objects) by specifying that only one thread should be allowed access to the shared object at any point in time. By allowing the flexibility of specifying the exact number of threads that can concurrently access shared data, they are ideal in situations where the amount of parallelism needs to be tightly controlled. Semaphores should be viewed as gates. As long as the gate is open, concurrent access to the shared data resource is allowed. Once the gate is shut, no more threads will be allowed access to the shared data resource.

Although similar to mutex objects, semaphores do not provide the deadlock checking facility provided by mutex objects. Acquisition of a semaphore does not result in disabling kernel-mode APCs. Note that storage for semaphore objects must be provided by your driver and should always be allocated from non-paged memory.

Here's how semaphores work. Each semaphore has an associated Count value. If the Count associated with the semaphore object is zero, any thread that waits for the semaphore object will be suspended. Whenever a thread that acquired the semaphore object releases the semaphore, the Count gets incremented by a specified amount (the Adjustment argument specified when releasing the semaphore). If incrementing the Count results in a transition from 0 to a nonzero value, then a certain number of waiting threads will have their wait satisfied.

Each time a wait is satisfied the Count gets decremented by 1; therefore, the number of waits that will get satisfied on a transition from 0 to a nonzero value will be equal to the value of the nonzero Count. The net result is that a fixed number of threads (bounded by the Limit value specified when initializing the semaphore) can concurrently acquire the semaphore and thereby concurrently access the shared resource.

The following routines are provided by the NT Kernel to support counting semaphore objects:

KeInitializeSemaphore()

You can specify the initial value of the Count associated with the semaphore. If the Count is nonzero, the semaphore will be set to the signaled state. You must also specify the maximum count that will be allowed for the semaphore. This Limit argument bounds the number of concurrent accesses to the shared data resource protected by the semaphore.

KeReleaseSemaphore{)

When releasing a semaphore, your driver can specify the Argument, which is the amount by which the Count associated with the semaphore should be incremented. This might result in satisfying one or more waiting threads. Note that if incrementing the count by the supplied Argument value results in exceeding the original Limit value (specified when initializing the semaphore), or if you specify a negative value for the Argument variable, the thread performing the release will encounter an exception of STATUS_SEMAPHORE_LIMIT_EXCEEDED.

KeReadStateSemaphore()

This routine returns the current value of the Count associated with the semaphore. This value should be interpreted as the number of waits that will be immediately satisfied for the semaphore object.

ERESOURCE Objects (Read/Write Locks)

The Windows NT Executive provides an important additional synchronization mechanism extensively used by file system drivers. The ERESOURCE structure is a primitive that provides single writer (exclusive access), multiple reader (shared access) semantics. Therefore, each thread has the flexibility of determining the type of access to request from the resource structure.

When a thread needs to modify the shared data protected by the resource, it must request the read/write lock exclusively. However, if the thread just needs to read the contents of the shared data protected by the resource, it will typically acquire the resource shared, allowing other threads to concurrently read the same shared data. If any thread acquires the resource exclusively, of course, no other thread can acquire it.

Storage for these read/write locks must be provided by the driver from nonpaged pool.

The ERESOURCE structure has the concept of an *owning thread* for the resource (multiple reader threads could concurrently own the same resource shared). Additionally, these read/write locks provide recursive acquisition functionality. However, the thread must release the lock as many times as it was acquired.

A note of caution: none of the dispatcher synchronization primitives discussed in this chapter needs to be uninitialized when a driver determines that the primitive is no longer needed and deallocates the memory reserved for the synchronization primitive. However, ERESOURCE structures must be uninitialized (or deleted from the global linked list of resources) before memory allocated for these structures can be deallocated.

Finally, all the resource manipulation routines require that the IRQL of the processor be less than or equal to DISPATCH_LEVEL.

NOTE The ERESOURCE structure uses an Executive spin lock to protect internal fields within the resource structure. When acquiring this spin lock, the Executive raises the IRQL for the processor to DISPATCH_ LEVEL. Therefore, invoking any of the routines at an IRQL greater than DISPATCH_ LEVEL could lead to a deadlock condition.

The following routines are provided by the NT Executive to support ERESOURCE structures:

`ExInitializeResourceLite()`

This simple routine initializes the resource structure allocated by the driver. The resource is added to a global linked list of resource structures, and therefore, it is important that the driver uninitialize the resource before freeing memory allocated to it.

`ExDeleteResourceLite()`

This routine unlinks the resource from the global linked list of resources. The memory reserved for this resource structure can subsequently be released.

`ExAcquireResourceExclusiveLite()`

This routine will attempt to acquire the resource structure exclusively (for write access). The thread requesting exclusive access can specify whether it wishes to wait (block) for the resource to become available. If the thread is not prepared to block, and if some other thread has the resource acquired shared or exclusively, this routine will return FALSE, indicating that the request to acquire the resource was unsuccessful.

`ExTryToAcquireResourceExclusiveLite()`

This routine is functionally equivalent to invoking `ExAcquireResourceExclusiveLite()` with the `Wait` argument set to FALSE. However, Microsoft literature claims that this call is more efficient.

`ExAcquireResourceSharedLite()`

This routine will attempt to acquire the resource structure shared (for read access). The thread requesting exclusive access can specify whether it wishes to wait (block) for the resource to become available. If the thread is not prepared to block and if some other thread has the resource acquired exclusively, this routine will return FALSE, indicating that the request to acquire the resource was unsuccessful. If other threads have this resource acquired shared, the current request for shared access will be successful and will return TRUE.

`ExReleaseResourceForThreadLite()`

Invoke this function to release a previously acquired resource. The thread ID (identifying the thread that is performing this operation) must be passed in as an argument to this routine. This thread ID can be obtained by a call to `ExGetCurrentResourceThread()`.

`ExAcquireSharedStarveExclusive()`

Typically, requests for resource acquisition are managed so that threads requesting exclusive access are not starved out. Starvation can occur under the following scenario:

A thread has the resource acquired shared. Subsequently, a request for exclusive acquisition arrives with the `Wait` argument set to TRUE. This request is therefore queued. Before the thread that has the resource shared releases the resource, another shared acquisition request is also received. If the NT Executive keeps satisfying the requests for shared acquisition while making the request for exclusive access wait, it is possible that the request for exclusive activation will get starved (i.e., will never be completed).

Therefore, the NT Executive will typically not satisfy a new request for shared access if a previous request for exclusive access is already queued.*

By using this call however, a thread deliberately requests that its request for shared access be given preference over any preexisting queued requests for exclusive access.

`ExAcquireSharedWaitForExclusive()`

This routine is the inverse of the previous one. Here, a shared access requester explicitly states that preference should be given to exclusive access requests even if such requests arrive after the current one. Therefore, the current request will only be satisfied if there are no pending exclusive requests for the resource (unless this is a recursive acquisition request).

Supporting Routines (RTLs)

The Windows NT Executive provides a substantial amount of support to kernel-mode driver developers via the run-time library and the filesystem run-time library.† These libraries should be explored thoroughly if you wish to develop kernel-mode drivers.

The run-time library consists of sets of routines that do the following:

- Manipulate doubly linked lists
- Query the Windows NT Registry and write information to the Registry
- Execute type conversion routines (character to string, etc.)
- Execute string manipulation routines for ASCII and Unicode strings (including conversion from ASCII to Unicode and vice versa)
- Copy, zero, move, fill, and compare memory blocks
- Perform 32-bit integer arithmetic and 64-bit large integer and long arithmetic (including conversion between types)

---

* Note that if a requesting thread already owns the resource exclusively and asks for shared access to the resource (recursively), the request is always granted.

† The file system run-time library (FSRTL) functions and structure headers are not declared in the DDK (although some of the RTL functions are exposed). However, throughout the course of this book, I will present important routines and structures defined in the file system run-time library. Microsoft released a Windows NT Installable File Systems (IFS) Developers Kit in April 1997. From all available information at the time of writing this book, the header files for structure definitions and function declarations contained within the FSRTL are only available as part of the Installable File Systems (IFS) kit from Microsoft for a sum of money in addition to the amount paid for the Device Driver's Kit (DDK).

- Perform time conversion and manipulation routines
- Create and manipulate security descriptors

Although routines contained in these two libraries are not discussed in detail here (see Chapter 2, *File System Driver Development*, for a discussion of some of them), example code throughout this book will use one or more of the functions, structure definitions, and macros contained within these libraries.

Run-time library functions can be easily identified by the prefix Rtl prepended to all function declarations. Similarly filesystem run-time library routines can be identified by the FsRtl prefix prepended to function declarations.

I highly recommend you familiarize yourself with the functionality provided by these two libraries, and that you use these routines in your driver whenever the need arises. You should use run-time library routines when you would have otherwise used standard C library routines, e.g., instead of using the memcpy ( ) library call, try to use the RtlCopyMemory () supporting routine. This will ensure correct behavior of your driver on all platforms.

Although header files for both of these libraries must be purchased from Microsoft as part of an IPS kit, this book will provide descriptions and sample usage of important structure definitions and function declarations provided by each of these libraries.