The ReadFile Example

User applications switch from user mode to kernel mode when they make a system service call. For example, a Windows *ReadFile* function eventually needs to call the internal Windows routine that actually handles reading data from a file. That routine, because it accesses internal system data structures, must run in kernel mode. The transition from user mode to kernel mode is accomplished by the use of a special processor instruction that causes the processor to switch to kernel mode. The operating system traps this instruction, notices that a system service is being requested, validates the arguments the thread passed to the system function, and then executes the internal function. Before returning control to the user thread, the processor mode is switched back to user mode. In this way, the operating system protects itself and its data from perusal and modification by user processes. Thus, it's normal for a user thread to spend part of its time executing in user mode and part in kernel mode. In fact, because the bulk of the graphics and windowing system also runs in kernel mode, graphics-intensive applications spend more of their time in kernel mode than in user mode.

On x86 processors prior to the Pentium II, Windows uses the int 0x2e instruction (46 decimal), which results in a trap. Windows fills in entry 46 in the IDT to point to the system service dispatcher. The trap causes the executing thread to transition into kernel mode and enter the system service dispatcher. A numeric argument passed in the EAX processor register indicates the system service number being requested. The EDX register points to the list of parameters the caller passes to the system service. To return to user mode, the system service dispatcher uses the iretd (interrupt return instruction).

On x86 Pentium II processors and higher, Windows uses the special sysenter instruction, which Intel defined specifically for fast system service dispatches. To support the instruction, Windows stores at boot time the address of the kernel's system service dispatcher routine in a machine specific register (MSR) associated with the instruction. The execution of the instruction causes the change to kernel mode and execution of the system service dispatcher. The system service number is passed in the EAX processor register and the EDX register points to the list of caller arguments. To return to user mode, the system service dispatcher usually executes the sysexit instruction. (In some cases, like when the single-step flag is enabled on the processor, the system service dispatcher uses the iretd instead because stepping over a sysexit instruction with the kernel debugger would result in an undefined system state leading to a crash.)

When an application calls a function in a subsystem DLL, one of three things can occur:

- The function is entirely implemented in user mode inside the subsystem DLL. In other words, no message is sent to the environment subsystem process, and no Windows executive system services are called. The function is performed in user mode, and the results are returned to the caller. Examples of such functions include *GetCurrentProcess* (which always returns –1, a value that is defined to refer to the current process in all process-related functions) and *GetCurrentProcessId*. (The process ID doesn't change for a running process, so this ID is retrieved from a cached location, thus avoiding the need to call into the kernel.)
- The function requires one or more calls to the Windows executive. For example, the Windows *ReadFile* and *WriteFile* functions involve calling the underlying internal (and undocumented) Windows I/O system services *NtReadFile* and *NtWriteFile*, respectively.
- The function requires some work to be done in the environment subsystem process. (The environment subsystem processes, running in user mode, are responsible for maintaining the state of the client applications running under their control.) In this case, a client/server request is made to the environment subsystem via a message sent to the subsystem to

perform some operation. The subsystem DLL then waits for a reply before returning to the caller.

I/O Operations

Some functions can be a combination of the second and third items just listed, such as the Windows *CreateProcess* and *CreateThread* functions. Most I/O operations that applications issue are *synchronous*; that is, the application thread waits while the device performs the data transfer and returns a status code when the I/O is complete. The program can then continue and access the transferred data immediately.

When used in their simplest form, the Windows *ReadFile* and *WriteFile* functions are executed aynchronously. They complete an I/O operation before returning control to the caller. *Asynchronous I/O* allows an application to issue an I/O request and then continue executing while the device transfers the data. This type of I/O can improve an application's throughput because it allows the application thread to continue with other work while an I/O operation is in progress. To use asynchronous I/O, you must specify the FILE_FLAG_OVERLAPPED flag when you call the Windows *CreateFile* function. Of course, after issuing an asynchronous I/O operation, the thread must be careful not to access any data from the I/O operation until the device driver has finished the data transfer. The thread must synchronize its execution with the completion of the I/O request by monitoring a handle of a synchronization object (whether that's an event object, an I/O completion port, or the file object itself) that will be signaled when the I/O is complete.

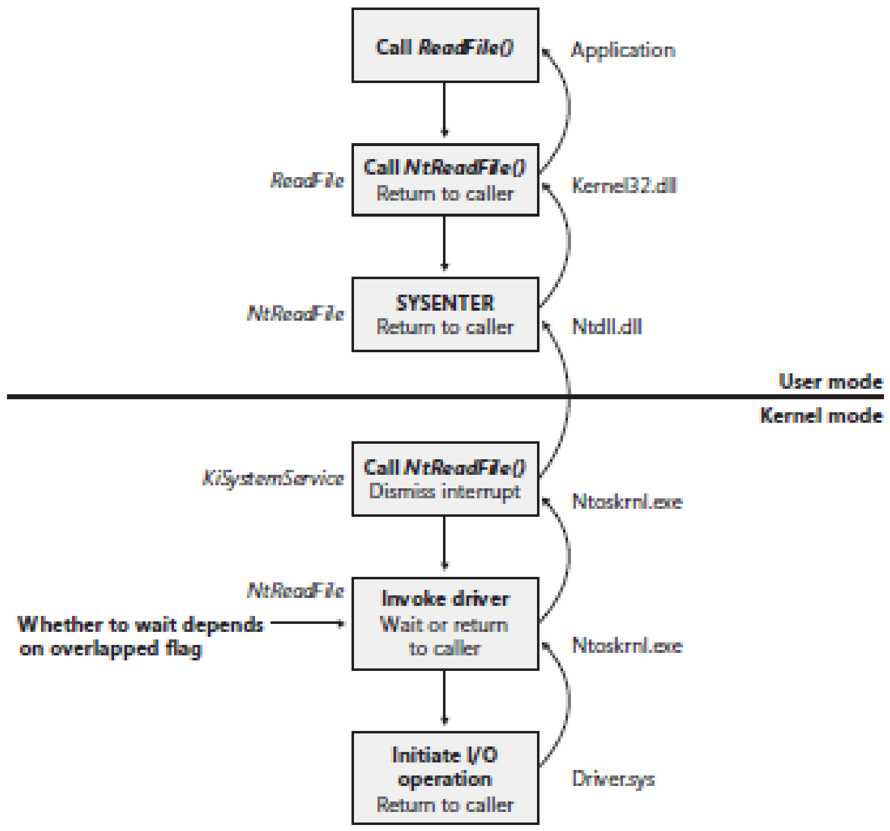
Call *ReadFile()*
Application
ReadFile
Call *NtReadFile()* Return to caller
Kernel32.dll
NtReadFile
SYSENTER Return to caller
Ntdll.dll
User mode
Kernel mode
KiSystemService
Call *NtReadFile()* Dismiss interrupt
Ntoskrnl.exe
NtReadFile
Whether to wait depends on overlapped flag
Invoke driver Wait or return to caller
Ntoskrnl.exe
Initiate I/O operation Return to caller
Driver.sys

FIGURE 7-8 Control flow for an I/O operation

Regardless of the type of I/O request, internally I/O operations issued to a driver on behalf of the application are performed asynchronously; that is, once an I/O request has been initiated, the device driver returns to the I/O system. Whether or not the I/O system returns immediately to the caller depends on whether the file was opened for synchronous or asynchronous I/O. Figure 7-8 illustrates the flow of control when a read operation is initiated. Notice that if a wait is done, which depends on the overlapped flag in the file object, it is done in kernel mode by the *NtReadFile* function.

The I/O request packet (IRP) is where the I/O system stores information it needs to process an I/O request. When a thread calls an I/O service, the I/O manager constructs an IRP to represent the operation as it progresses through the I/O system. When an application or a device driver indirectly creates an IRP by using the NtReadFile, NtWriteFile, or NtDeviceIoControlFile system services (or the Windows API functions corresponding to these services, which are ReadFile, WriteFile, and DeviceIoControl), the I/O manager determines whether it needs to participate in the management of the caller's input or output buffers.

Other Key Functions

Omitted from this description is the role of other Executive components in the Ntoskrnl.exe. A file is an object, and while the I/O Manager and the Filesystem Driver actually read the file, several other Executive Modules play a role:

- The Object Manager creates the file object data structure.
- The Security Reference Monitor ensures that the user application Access Token is valid by checking the files Access Control List.
- The Virtual Memory Manager uses the file object data to load the file into virtual memory.
- The Cache Manager (nothing to do with L1-L2 cache) caches the file in RAM for fast access.

The application ultimately reads the file in RAM. The following diagrams indicate the central actors in the Executive.

Asynchronous I/O Cancellation

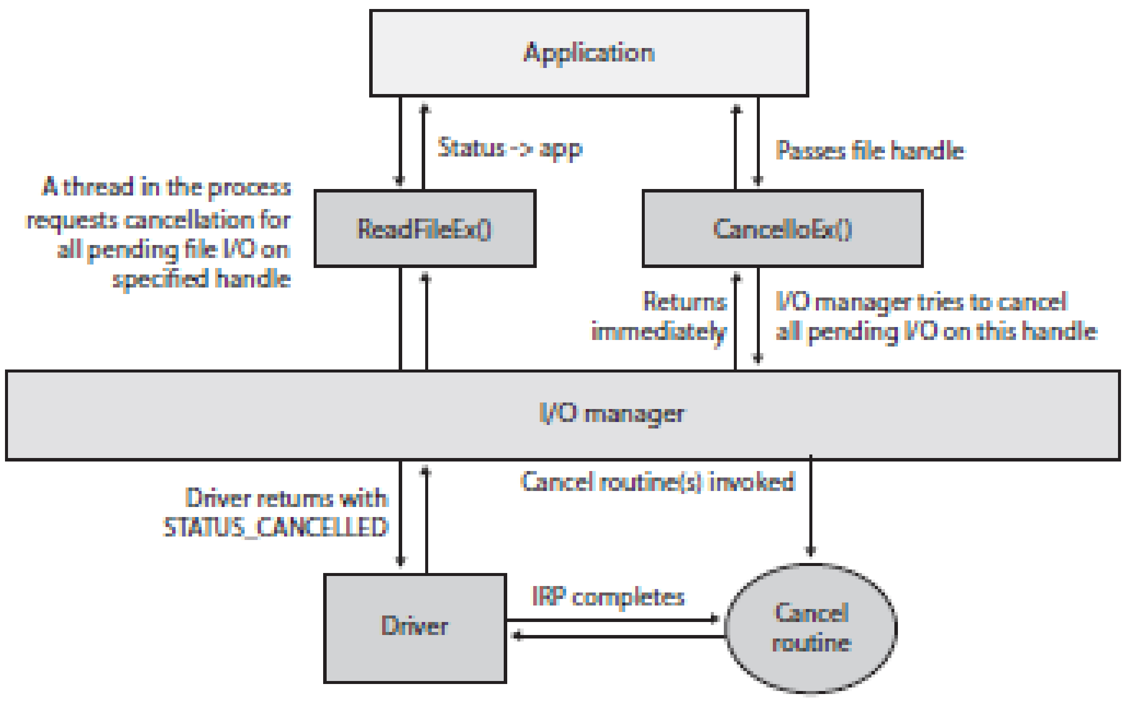
Application
Status -> app
Passes file handle
A thread in the process requests cancellation for all pending file I/O on specified handle
ReadFileEx()
CancelIoEx()
Returns immediately
I/O manager tries to cancel all pending I/O on this handle
I/O manager
Driver returns with STATUS_CANCELLED
Cancel routine(s) invoked
Driver
IRP completes
Cancel routine

FIGURE 7-21 Asynchronous I/O cancellation

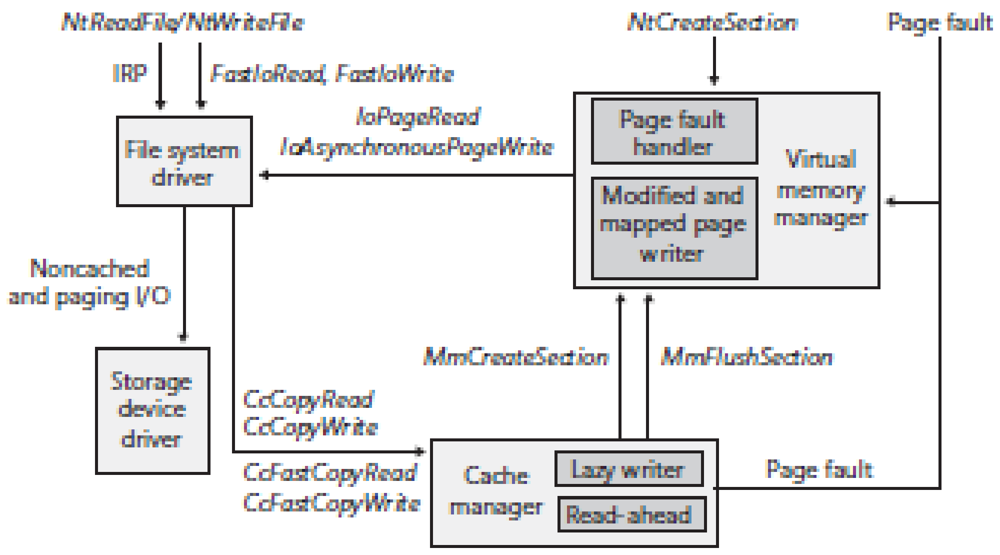

FIGURE 10-10 File system interaction with cache and memory managers