Interrupt Request Level

Windows NT assigns a priority level known as the *interrupt request level* to each hardware interrupt and to a select few software events. IRQLs provide a synchronization method for activities on a single CPU based on the following rule:

> **Once a CPU is executing at an IRQL above PASSIVE_LEVEL, an activity on that CPU can be preempted only by an activity that executes at a higher IRQL.**

Figure 4-1 illustrates the range of IRQL values for the x86 platform. (In general, the numeric values of IRQL depend on which platform you're talking about.) User-mode programs execute at PASSIVE_LEVEL and are therefore preemptable by any activity that executes at an elevated IRQL. Many of the functions in a device driver also execute at PASSIVE_LEVEL. The **DriverEntry** and **AddDevice** routines discussed in Chapter 2, "Basic Structure of a WDM Driver," are in this category, as are most of the I/O request packet (IRP) dispatch routines that I'll discuss in ensuing chapters.

Certain common driver routines execute at DISPATCH_LEVEL, which is higher than PASSIVE_LEVEL. These include the **StartIo** routine, deferred procedure call (DPC) routines, and many others. What they have in common is a need to access fields in the device object and device extension without interference from driver dispatch routines and each other. When one of these routines is running, the rule stated earlier guarantees that no thread can preempt it to execute a driver dispatch routine because the dispatch routine runs at a lower IRQL. Furthermore, no thread could preempt it to run another of these special routines because that other routine would run at the same IRQL. The rule, once again, is that preemption is allowed to run only an activity at a *higher* IRQL.

> **NOTE**
>
> *Dispatch routine* and *DISPATCH_LEVEL* have unfortunately similar names. Dispatch routines are so called because the I/O Manager dispatches I/O requests to them. DISPATCH_LEVEL is so called because it's the IRQL at which the kernel's thread dispatcher originally ran when deciding which thread to run next. (The thread dispatcher now usually runs at SYNCH_LEVEL, if you care.)

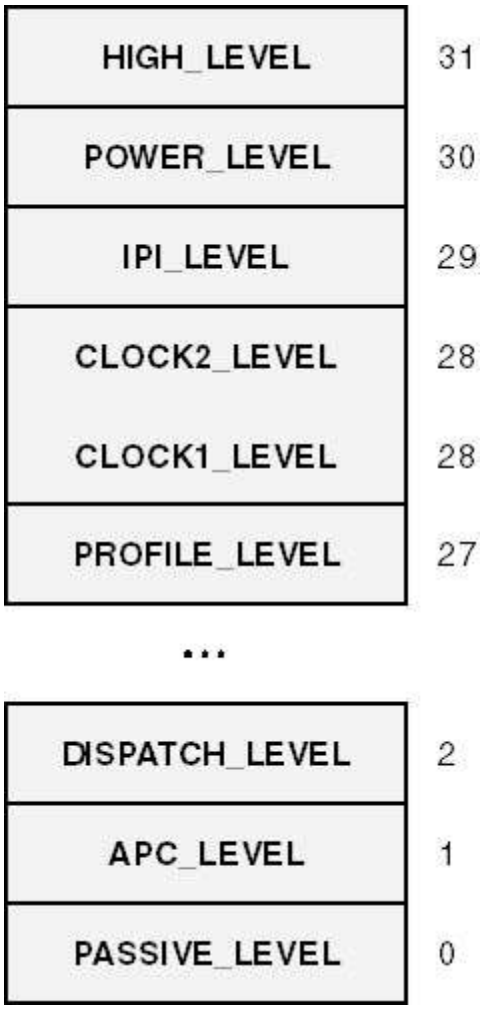

Figure 4-1. *Interrupt request levels.*

Between DISPATCH_LEVEL and PROFILE_LEVEL is room for various hardware interrupt levels. In general, each device that generates interrupts has an IRQL that defines its interrupt priority vis-à-vis other devices. A WDM driver discovers the IRQL for its interrupt when it receives an IRP_MJ_PNP request with the minor function code IRP_MN_START_DEVICE. The device's interrupt level is one of the many items of configuration information passed as a parameter to this request. We often refer to this level as the *device IRQL*, or DIRQL for short. DIRQL is not a single request level. Rather, it is the IRQL for the interrupt associated with whichever device is under discussion at the time.

The other IRQL levels have meanings that sometimes depend on the particular CPU architecture. Since those levels are used internally by the Windows NT kernel, their meanings aren't especially germane to the job of writing a device driver. The purpose of APC_LEVEL, for example, is to allow the system to schedule an *asynchronous procedure call* (APC), which I'll describe in detail later in this chapter, for a particular thread without interference from some other thread on the same CPU. Operations that occur at HIGH_LEVEL include taking a memory snapshot just prior to hibernating the computer, processing a bug check, handling a totally spurious interrupt, and others. I'm not going to attempt to provide an exhaustive list here because, as I said, you and I don't really need to know all the details.

IRQL in Operation

To illustrate the importance of IRQL, refer to Figure 4-2, which illustrates a possible time sequence of events on a single CPU. At the beginning of the sequence, the CPU is executing at PASSIVE_LEVEL. At time t_1, an interrupt arrives whose service routine executes at IRQL-1,

one of the levels between DISPATCH_LEVEL and PROFILE_LEVEL. Then, at time t_2, another interrupt arrives whose service routine executes at IRQL-2, which is less than IRQL-1. Because of the preemption rule already discussed, the CPU continues servicing the first interrupt. When the first interrupt service routine completes at time t_3, it might request a DPC. DPC routines execute at DISPATCH_LEVEL. Consequently, the highest priority pending activity is the service routine for the second interrupt, which therefore executes next. When it finishes at t_4, assuming nothing else has occurred in the meantime, the DPC will run at DISPATCH_LEVEL. When the DPC routine finishes at t_5, IRQL can drop back to PASSIVE_LEVEL.

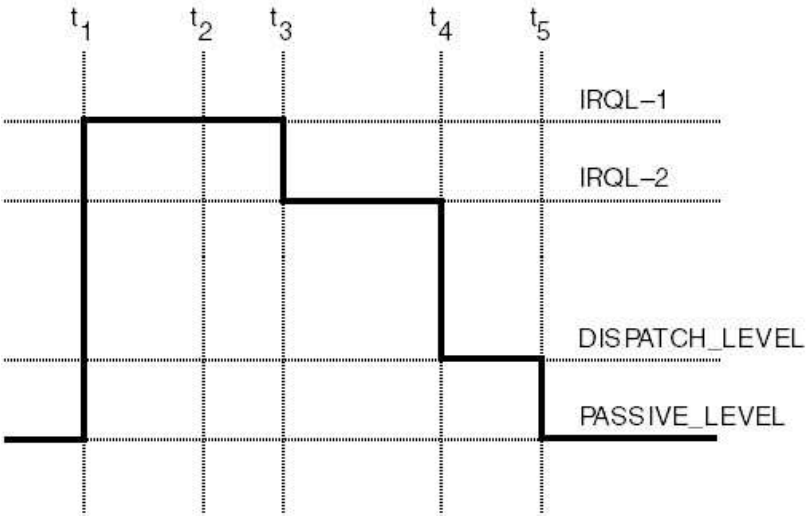

Figure 4-2. *Interrupt priority in action.*

The Basic Synchronization Rule

You can take advantage of IRQL's synchronizing effects by following this rule:

> **Always access shared data at the same elevated IRQL.**

In other words, whenever and wherever your code will access a data object that it shares with some other code, make sure that you execute at some specified IRQL above PASSIVE_LEVEL. Once above PASSIVE_LEVEL, the operating system won't allow preemption by another activity at the same IRQL, so you thereby forestall potential interference. Following this rule isn't sufficient to protect data on a multiprocessor machine, however, so you often need to take the additional precaution of acquiring a spin lock, as described in "Spin Locks" later in this chapter. If you only had to worry about operations on a single CPU, IRQL might be the only synchronizing concept you'd need to use, but the reality is that all WDM drivers must be designed to run on multiprocessor systems.

IRQL Compared with Thread Priorities

Thread priority is a very different concept than IRQL. Thread priority controls the actions of the scheduler in deciding when to preempt running threads and what thread to start running next. No thread switching occurs at or above DISPATCH_LEVEL, however. Whatever thread is active at the time IRQL rises to DISPATCH_LEVEL remains active at least until IRQL drops below DISPATCH_LEVEL. The only "priority" that means anything at elevated IRQL is IRQL itself, and it controls which programs can execute rather than the thread context within which they execute.

IRQL and Paging

One consequence of running at elevated IRQL is that the system becomes incapable of servicing page faults. The rule this fact implies is simply stated:

Code executing at or above DISPATCH_LEVEL must not cause page faults.

One implication of this rule is that any of the subroutines in your driver that execute at or above DISPATCH_LEVEL must be in nonpaged memory. Furthermore, all the data you access in such a subroutine must also be in nonpaged memory. Finally, as IRQL rises, fewer and fewer kernel-mode support routines are available for your use.

The DDK documentation explicitly states the IRQL restrictions on support routines. For example, the entry for **KeWaitForSingleObject** indicates two restrictions:

1. The caller must be running at or below DISPATCH_LEVEL.
2. If a nonzero timeout period is specified in the call, the caller must be running strictly below DISPATCH_LEVEL.

Reading between the lines, what is being said here is this: if the call to KeWaitForSingleObject might conceivably block for any period of time (that is, you've specified a nonzero timeout), you must be below DISPATCH_LEVEL, where thread blocking is permitted. If all you want to do is check to see if an event has been signalled, however, you can be at DISPATCH_LEVEL. You cannot call this routine at all from an interrupt service routine or other routine running above DISPATCH_LEVEL.

Implicitly Controlling IRQL

Most of the time, the system calls the routines in your driver at the correct IRQL for the activities you're supposed to carry out. Although I haven't discussed many of these routines in detail, I want to give you an example of what I mean. Your first encounter with a new I/O request is when the I/O Manager calls one of your dispatch routines to process an IRP. The call occurs at PASSIVE_LEVEL because you might need to block the calling thread and you might need to call any support routine at all. You can't block a thread at a higher IRQL, of course, and PASSIVE_LEVEL is the only level at which there are no restrictions on the support routines you can call.

If your dispatch routine queues the IRP by calling **IoStartPacket** , your next encounter with the request will be when the I/O Manager calls your StartIo routine. This call occurs at DISPATCH_LEVEL because the system needs to access the queue of I/O requests without interference from the other routines that are inserting and removing IRPs from the queue. Remember the rule stated earlier: always access shared data objects at the same (elevated) IRQL. Since every routine that accesses the IRP queue does so at DISPATCH_LEVEL, it's not possible (on a single CPU, that is) for anyone to be interrupted in the middle of an operation on the queue.

Later on, your device might generate an interrupt, whereupon your interrupt service routine will be called at DIRQL. It's likely that some registers in your device can't safely be shared. If you only access those registers at DIRQL, you can be sure that no one can interfere with your interrupt service routine (ISR) on a single-CPU computer. If other parts of your driver need to access these crucial hardware registers, you would guarantee that those other parts execute only at DIRQL. The **KeSynchronizeExecution** service function helps you enforce that rule, and I'll discuss it in Chapter 7, "Reading and Writing Data," in connection with interrupt handling.

Still later, you might arrange to have a DPC routine called. DPC routines execute at DISPATCH_LEVEL because, among other things, they need to access your IRP queue to remove the next request from a queue and pass it to your StartIo routine. You call the **IoStartNextPacket** service routine to extract the next request from the queue, and it must be called at DISPATCH_LEVEL. It might call your StartIo routine before returning. Notice how neatly the IRQL requirements dovetail here: queue access, the call to IoStartNextPacket, and the possible call to StartIo are all required to occur at DISPATCH_LEVEL, and that's the level at which the system calls the DPC routine.

Although it's possible for you to explicitly control IRQL (and I'll explain how in the next section), there's seldom any reason to do so because of the correspondence between your needs and the level at which the system calls you. Consequently, you don't need to get hung up on which IRQL you're executing at from moment to moment: it's almost surely the correct level for the work you're supposed to do right then.

Explicitly Controlling IRQL

When necessary, you can raise and subsequently lower the IRQL on the current processor by calling **KeRaiseIrql** and **KeLowerIrql**. For example, from within a routine running at PASSIVE_LEVEL:

```
1→   KIRQL oldirql;
2→   ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
3→   KeRaiseIrql(DISPATCH_LEVEL, &oldirql);
     ...
4→   KeLowerIrql(oldirql);
```

1. KIRQL is the typedef name for an integer that holds an IRQL value. We'll need a variable to hold the current IRQL, so we declare it this way.

2. This ASSERT expresses a necessary condition for calling KeRaiseIrql: the new IRQL must be greater than or equal to the current level. If this relation isn't true, KeRaiseIrql will bugcheck (that is, report a fatal error via a blue screen of death).

3. KeRaiseIrql raises the current IRQL to the level specified by the first argument. It also saves the current IRQL at the location pointed to by the second argument. In this example, we're raising IRQL to DISPATCH_LEVEL and saving the current level in **oldirql**.

4. After executing whatever code we desired to execute at elevated IRQL, we lower the request level back to its previous value by calling KeLowerIrql and specifying the oldirql value previously returned by KeRaiseIrql.

The DDK documentation says that you must call KeLowerIrql with the same value returned by the immediately preceding call to KeRaiseIrql. This is true in the larger sense that you *should* restore IRQL to what it was before you raised it. Otherwise, various assumptions made by code you call later or by the code which called you can later turn out to be incorrect. This statement in the documentation isn't true in the exact sense, however, because the only rule that KeLowerIrql actually applies is that the new IRQL must be less than or equal to the current one.

It's a mistake (and a big one!) to lower IRQL below whatever it was when some system routine called your driver, even if you raise it back before returning. Such a break in synchronization might allow some activity to preempt you and interfere with a data object that your caller assumed would remain inviolate.

You can use a special routine if you want to raise the IRQL to DISPATCH_LEVEL:

```
KIRQL oldirql = KeRaiseIrqlToDpcLevel();
...
KeLowerIrql(oldirql);
```

The advantage of using this service call is that you don't need to know or remember that DISPATCH_LEVEL is the level you're aiming for. In addition, since **KeRaiseIrqlToDpcLevel** returns the current IRQL as its value, this function is slightly more convenient to use than KeRaiseIrql.