Asynchronous Procedure Calls

An *asynchronous procedure call* (APC) is a function that executes asynchronously in the context of a particular thread. When an APC is queued to a thread, the system issues a software interrupt. The next time the thread is scheduled, it will run the APC function. An APC generated by the system is called a *kernel-mode APC*. An APC generated by an application is called a *user-mode APC*. A thread must be in an alertable state to run a user-mode APC.

Each thread has its own APC queue. An application queues an APC to a thread by calling the **QueueUserAPC** function. The calling thread specifies the address of an APC function in the call to **QueueUserAPC**. The queuing of an APC is a request for the thread to call the APC function.

When a user-mode APC is queued, the thread to which it is queued is not directed to call the APC function unless it is in an alertable state. A thread enters an alertable state when it calls the **SleepEx**, **SignalObjectAndWait**, **MsgWaitForMultipleObjectsEx**, **WaitForMultipleObjectsEx**, or **WaitForSingleObjectEx** function. If the wait is satisfied before the APC is queued, the thread is no longer in an alertable wait state so the APC function will not be executed. However, the APC is still queued, so the APC function will be executed when the thread calls another alertable wait function.

The **ReadFileEx**, **SetWaitableTimer**, **SetWaitableTimerEx**, and **WriteFileEx** functions are implemented using an APC as the completion notification callback mechanism.

If you are using a **thread pool**, note that APCs do not work as well as other signaling mechanisms because the system controls the lifetime of thread pool threads, so it is possible for a thread to be terminated before the notification is delivered. Instead of using an APC-based signaling mechanism such as the *pfnCompletionRoutine* parameter of **SetWaitableTimer** or **SetWaitableTimerEx**, use a waitable object such as a timer created with **CreateThreadpoolTimer**. For I/O, use an I/O completion object created with **CreateThreadpoolIo** or an *hEvent*-based **OVERLAPPED** structure where the event can be passed to the **SetThreadpoolWait** function.

Synchronization Internals

When an I/O request is issued, a structure is allocated to represent the request. This structure is called an I/O request packet (IRP). With synchronous I/O, the thread builds the IRP, sends it to the device stack, and waits in the kernel for the IRP to complete. With asynchronous I/O, the thread builds the IRP and sends it to the device stack. The stack might complete the IRP immediately, or it might return a pending status indicating that the request is in progress. When this happens, the IRP is still associated with the thread, so it will be canceled if the thread terminates or calls a function such as **CancelIo**. In the meantime, the thread can continue to perform other tasks while the device stack continues to process the IRP.

There are several ways that the system can indicate that the IRP has completed:

- Update the overlapped structure with the result of the operation so the thread can poll to determine whether

the operation has completed.

- Signal the event in the overlapped structure so a thread can synchronize on the event and be woken when the operation completes.
- Queue the IRP to the thread's pending APC so that the thread will execute the APC routine when it enters an alertable wait state and return from the wait operation with a status indicating that it executed one or more APC routines.
- Queue the IRP to an I/O completion port, where it will be executed by the next thread that waits on the completion port.

Threads that wait on an I/O completion port do not wait in an alertable state. Therefore, if those threads issue IRPs that are set to complete as APCs to the thread, those IPC completions will not occur in a timely manner; they will occur only if the thread picks up a request from the I/O completion port and then happens to enter an alertable wait.

Related topics

Using a Waitable Timer with an Asynchronous Procedure Call

© 2016 Microsoft