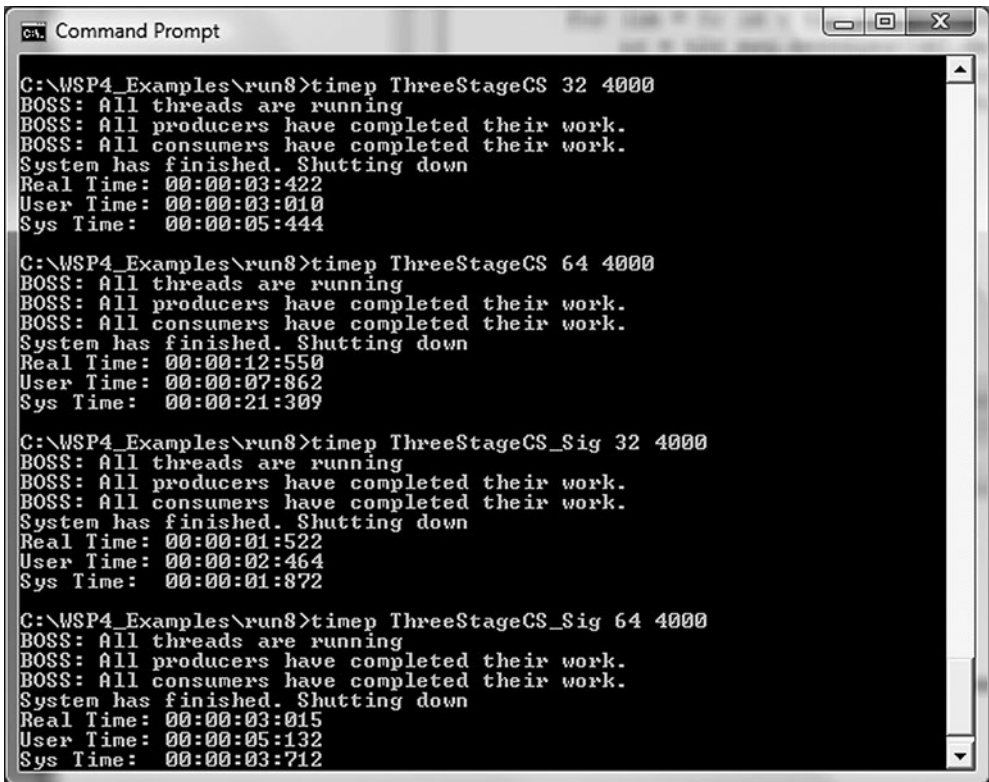


- `ThreeStage_noSOAW` does not use `SignalObjectAndWait`; instead it uses successive mutex and event waits with a time-out. This corresponds to the code fragment at the beginning of the chapter.
- `ThreeStage_Sig` uses the signal model (auto-reset/`SetEvent`) with `SignalObjectAndWait` and will work if only one message is produced at a time, as is the case in this example. There are significant performance advantages because only a single thread is released to test the predicate.
- `ThreeStageCS_Sig` is like `ThreeStage_Sig`, except that it uses a CS in place of a mutex. It combines the features of `ThreeStageCS` and `ThreeStage_Sig`.
- `ThreeStageSig_noSOAW` combines the `ThreeStage_noSOAW` and `ThreeStage_Sig` features.
- `ThreeStageCV` uses Windows NT6 condition variables, which are described later in the chapter.



```

C:\WSP4_Examples\run8>tinep ThreeStageCS 32 4000
BOSS: All threads are running
BOSS: All producers have completed their work.
BOSS: All consumers have completed their work.
System has finished. Shutting down
Real Time: 00:00:03:422
User Time: 00:00:03:010
Sys Time: 00:00:05:444

C:\WSP4_Examples\run8>tinep ThreeStageCS 64 4000
BOSS: All threads are running
BOSS: All producers have completed their work.
BOSS: All consumers have completed their work.
System has finished. Shutting down
Real Time: 00:00:12:550
User Time: 00:00:07:862
Sys Time: 00:00:21:309

C:\WSP4_Examples\run8>tinep ThreeStageCS_Sig 32 4000
BOSS: All threads are running
BOSS: All producers have completed their work.
BOSS: All consumers have completed their work.
System has finished. Shutting down
Real Time: 00:00:01:522
User Time: 00:00:02:464
Sys Time: 00:00:01:872

C:\WSP4_Examples\run8>tinep ThreeStageCS_Sig 64 4000
BOSS: All threads are running
BOSS: All producers have completed their work.
BOSS: All consumers have completed their work.
System has finished. Shutting down
Real Time: 00:00:03:015
User Time: 00:00:05:132
Sys Time: 00:00:03:712

```

Run 10-5b `ThreeStageCS[_Sig]: CS Broadcast and Signaling`

Appendix C also shows the comparative performance of these implementations; the run screen shots here show some initial results.

Run 10–5a compares `ThreeStage` and `ThreeStage_Sig` with 32 and 64 producer/consumer pairs. Both use a mutex, but `ThreeStage`, which broadcasts, performs poorly and does not scale as we go from 32 to 64 threads.

Run 10–5b makes the same comparison, but with a `CRITICAL_SECTION` and a time-out in the consumer loop. CS performance is much better, as expected, but, again, the broadcast model does not scale well with the number of producer/consumer pairs.

Windows NT6 Condition Variables

Windows Vista and 2008 Server support condition variable objects whose behavior is similar to Pthreads condition variables and the CV model we’ve used in this chapter. Furthermore, Windows condition variables (WCV, a nonstandard but convenient abbreviation) use `CRITICAL_SECTION` and SRW lock objects (Chapter 8) rather than mutexes, and the WCV objects are also user, not kernel, objects, providing additional performance benefits. The only significant limitations are:

- Condition variables cannot be shared between processes the way you can share named mutexes and events.
- There is nothing comparable to `SignalObjectAndWait`’s alertable state (see the upcoming “Asynchronous Procedure Calls” section), so you cannot cancel threads waiting on condition variables.

First, the type for a WCV object is `CONDITION_VARIABLE`. Initialize WCVs just as you would a `CRITICAL_SECTION` with the `InitializeConditionVariable` function. There is no `DeleteConditionVariable` function analogous to `DeleteCriticalSection` for the same reason that there is no delete function for SRW locks.

```
VOID InitializeConditionVariable (
    PCONDITION_VARIABLE pConditionVariable)
```

Use `SleepConditionVariableCS` with a `CRITICAL_SECTION` to wait for a signal to a WCV. Be sure to initialize both the CS and the WCV before their first use. There is a time-out, and the function looks similar to `SignalObjectAndWait`, except there is no alertable flag.

```

BOOL WINAPI SleepConditionVariableCS (
    PCONDITION_VARIABLE pConditionVariable,
    PCRITICAL_SECTION pCriticalSection,
    DWORD dwMilliseconds)

```

`SleepConditionVariableSRW` is an alternative, using SRW locks. The parameters are the same as for `SleepConditionVariableCS`, except there is an additional parameter to indicate whether the SRW lock is in shared or exclusive mode.

Signal, or “wake up,” a condition variable with `WakeConditionVariable` (corresponding to the “signal” model) and `WakeAllConditionVariable` (corresponding to the “broadcast” model).

Revising `QueueObject` (Program 10–4) is simple. First, modify `SynchObj.h` (Program 10–3) by replacing the three `HANDLE` items with `CRITICAL_SECTION` (for `qGuard`) and `CONDITION_VARIABLE` (for `qNf` and `qNe`). Then, `QueueObjectCV` is simpler, as there is no need for the additional wait after the SOAW call, as Program 10–6 shows. Note that there is no need to modify utility functions such as `QueueEmpty` and `QueueRemove`.

Program 10–6 implements the signal, rather than the broadcast, version. It also uses a CS, but the *Examples* file version uses an SRW lock, so there are illustrations of both techniques. MSDN’s example code (search for “Using Condition Variables”) is very similar, also using queues with “not empty” and “not full” predicates.

Program 10–6 QueueObjCV: The Queue Management Functions

```

/* Chapter 10. QueueObjCV. */
/* Queue functions - Using Windows NT6 Condition Variables*/

#include "Everything.h"
#include "SynchObj.h"

/* Finite bounded queue management functions. */

DWORD QueueGet (QUEUE_OBJECT *q, PVOID msg, DWORD mSize,
               DWORD maxWait)
{
    EnterCriticalSection (&q->qGuard);
    while (QueueEmpty (q)) {
        SleepConditionVariableCS (&q->qNe, &q->qGuard, INFINITE);
    }
}

```

```

    /* remove the message from the queue */
    QueueRemove (q, msg, mSize);

    /* Signal that the queue is not full as we've removed a message */
    WakeConditionVariable (&q->qNf);
    LeaveCriticalSection (&q->qGuard);
    return 0;
}

DWORD QueuePut (QUEUE_OBJECT *q, PVOID msg, DWORD mSize,
                DWORD maxWait)
{
    EnterCriticalSection (&q->qGuard);
    while (QueueFull (q)) {
        SleepConditionVariableCS(&q->qNf, &q->qGuard, INFINITE)
    }
    /* Put the message in the queue */
    QueueInsert (q, msg, mSize);

    /* Signal that the queue is not empty; we've inserted a message */
    WakeConditionVariable (&q->qNe);
    LeaveCriticalSection (&q->qGuard);
    return 0;
}

DWORD QueueInitialize (QUEUE_OBJECT *q, DWORD mSize, DWORD nMsgs)
{
    /* Initialize queue, including its mutex and events */
    /* Allocate storage for all messages. */

    q->qFirst = q->qLast = 0;
    q->qSize = nMsgs;

    InitializeCriticalSection(&q->qGuard);
    InitializeConditionVariable(&q->qNe);
    InitializeConditionVariable(&q->qNf);

    if ((q->msgArray = calloc (nMsgs, mSize)) == NULL) return 1;
    return 0; /* No error */
}

DWORD QueueDestroy (QUEUE_OBJECT *q)
{
    /* Free all the resources created by QueueInitialize */
    EnterCriticalSection (&q->qGuard);
    free (q->msgArray);
    LeaveCriticalSection (&(q->qGuard));
    DeleteCriticalSection (&(q->qGuard));
    return 0;
}

```

```

DWORD QueueEmpty (QUEUE_OBJECT *q)
{
    return (q->qFirst == q->qLast);
}

DWORD QueueFull (QUEUE_OBJECT *q)
{
    return ((q->qFirst - q->qLast) == 1 ||
            (q->qLast == q->qSize-1 && q->qFirst == 0));
}

DWORD QueueRemove (QUEUE_OBJECT *q, PVOID msg, DWORD mSize)
{
    char *pm;

    if (QueueEmpty(q)) return 1; /* Error - Q is empty */
    pm = q->msgArray;
    /* Remove oldest ("first") message */
    memcpy (msg, pm + (q->qFirst * mSize), mSize);
    q->qFirst = ((q->qFirst + 1) % q->qSize);
    return 0; /* no error */
}

DWORD QueueInsert (QUEUE_OBJECT *q, PVOID msg, DWORD mSize)
{
    char *pm;

    if (QueueFull(q)) return 1; /* Error - Q is full */
    pm = q->msgArray;
    /* Add a new youngest ("last") message */
    memcpy (pm + (q->qLast * mSize), msg, mSize);
    q->qLast = ((q->qLast + 1) % q->qSize);
    return 0;
}

```

The modified solution, in the *Examples* file, is *ThreeStageCV*, and it does provide the anticipated performance improvements relative to *ThreeStageCS* (the *CRITICAL_SECTION* solution), as shown in Run 10-6.

```

C:\WSP4_Examples\run8>timep ThreeStageCS 64 4000
BOSS: All threads are running
BOSS: All producers have completed their work.
BOSS: All consumers have completed their work.
System has finished. Shutting down
Real Time: 00:00:13:018
User Time: 00:00:07:222
Sys Time: 00:00:23:509

C:\WSP4_Examples\run8>timep ThreeStageCS 128 4000
BOSS: All threads are running
BOSS: All producers have completed their work.
BOSS: All consumers have completed their work.
System has finished. Shutting down
Real Time: 00:00:50:947
User Time: 00:00:23:602
Sys Time: 00:01:25:457

C:\WSP4_Examples\run8>timep ThreeStageCV 64 4000
BOSS: All threads are running
BOSS: All producers have completed their work.
BOSS: All consumers have completed their work.
System has finished. Shutting down
Real Time: 00:00:02:507
User Time: 00:00:06:021
Sys Time: 00:00:02:776

C:\WSP4_Examples\run8>timep ThreeStageCV 128 4000
BOSS: All threads are running
BOSS: All producers have completed their work.
BOSS: All consumers have completed their work.
System has finished. Shutting down
Real Time: 00:00:05:740
User Time: 00:00:12:636
Sys Time: 00:00:07:659

```

Run 10-6 ThreeStageCV: Condition Variable and CS Performance

Asynchronous Procedure Calls

A complexity in `ThreeStage` (Program 10-5), as it is currently written, is the way that the transmitter and receiver threads test the message sequence numbers and track the number of active consumers. This solution assumes that the transmitter and receiver threads know the number of consumers and understand the message structure, which may not always be the case. In general, it would be convenient if the boss thread were able to cancel the transmitter and receiver threads directly.

Another open problem is that there is no general method (other than `TerminateThread`) to signal, or cause an action in, a specific thread. Events signal one thread waiting on an auto-reset event or all the threads waiting on a manual-reset event, but there is no way to assure that the signal goes to a particular thread. The solution used so far is simply to wake up all the waiting threads so they can individually determine whether it is time to proceed. An alternative solution, oc-

asionally used, is to assign events to specific threads so that the signaling thread can determine which event to pulse or set.

APCs provide a solution to both of these problems. The sequence of actions is as follows, where the boss thread needs to control a cooperating worker or target thread.

- The boss thread specifies an APC callback routine to be executed by the target thread by queuing the APC to the target. More than one APC can be queued to a specific thread.
- The target thread enters an *alertable* wait state indicating that the thread can safely execute the APC. The order of these first two steps is irrelevant, so there is no concern here with race conditions.
- A thread in an alertable wait state will execute all queued APCs, one at a time.
- An APC can carry out any appropriate action, such as freeing resources or raising an exception. In this way, the boss thread can cause an exception to occur in the target, although the exception will not occur until the target has entered an alertable state.

APC execution is asynchronous in the sense that a boss thread can queue an APC to a target at any time, but the execution is synchronous in the sense that it can occur only when the target thread allows it to occur by entering an alertable wait state. Also notice that APCs give a limited sort of thread pool (see Chapter 9); the target thread is the “pool,” and the queued functions are the callback functions.

Alertable wait states appear once more in Chapter 14, which covers asynchronous I/O.

The following sections describe the required functions and illustrate their use with another variation of the `ThreeStage` program. In the *Examples* file, the source file is `ThreeStageCancel.c`, and the project to build this version is `ThreeStageCancel`.

Queuing Asynchronous Procedure Calls

One thread (the boss) queues an APC to a target thread using `QueueUserAPC`.

```

DWORD QueueUserAPC (
    PAPCFUNC pfnAPC,
    HANDLE hThread,
    ULONG_PTR dwData)

```

`pfnAPC` is a pointer to the actual function that the target thread will execute. `hThread` is the handle of the target thread. `dwData` is a pointer-sized argument value that will be passed to the APC function when it is executed.

`ThreeStageCancel.c`, in the main function (compare to Program 10–5), uses `QueueUserAPC` calls to cancel the transmitter and receiver threads after the consumer and producer threads terminate, as follows:

```

tstatus = QueueUserAPC
    (QueueShutDown, transmitterThread, 1);
if (tstatus == 0) ReportError (
    "Failed queuing APC for transmitter", 8, FALSE);
tstatus = QueueUserAPC
    (QueueShutDown, receiverThread, 2);
if (tstatus == 0) ReportError (...);

```

The `QueueUserAPC` return value is nonzero for success or zero for failure. `GetLastError()`, however, does not return a useful value, so the `ReportError` call does not request an error message (the last argument is `FALSE`).

`QueueShutDown` is an additional queue function, where the argument specifies shutting down the get queue (value 1) or the put queue (value 2). The function also sets flags that `QueueGet` and `QueuePut` test, so an APC queued by some other thread will not inadvertently shut down the queue.

Program 10–7 shows `QueueShutDown` working with modified versions of `QueueGet` and `QueuePut` (Program 10–4). As a result, the queue functions return nonzero values, causing the transmitter and receiver threads to unblock and exit.

Alertable Wait States

The last `SignalObjectAndWait` parameter, `bAlertable`, has been `FALSE` in previous examples. By using `TRUE` instead, we indicate that the wait is a so-called alertable wait, and the thread enters an alertable wait state. The behavior is as follows.

- If one or more APCs are queued to the thread (as a `QueueUserAPC` target thread) *before* either `hObjectToWaitOn` (normally an event) is signaled or the time-out expires, then the APCs are executed (there is no guaranteed order) and `SignalObjectAndWait` returns with a return value of `WAIT_IO_COMPLETION`.
- If an APC is never queued, then `SignalObjectAndWait` behaves in the normal way; that is, it waits for the object to be signaled or the time-out period to expire.

Alterable wait states will be used again with asynchronous I/O (Chapter 14); the name `WAIT_IO_COMPLETION` comes from this usage. A thread can also enter an alertable wait state with other alertable wait functions such as `WaitForSingleObjectEx`, `WaitForMultipleObjectsEx`, and `SleepEx`, and these functions will also be useful when performing asynchronous I/O.

We can now modify `QueueGet` and `QueuePut` (see Program 10–4) to perform an orderly shutdown after an APC is performed, even though the APC function, `QueueShutDown`, does not do anything other than print a message and return. All that is required is to enter an alertable wait state and to test the `SignalObjectAndWait` return value, as shown by the following modified queue functions (see `QueueObjCancel.c` in the *Examples* file).

This version uses the signal CV model with an auto-reset event and `SetEvent`; there is no need to be concerned with the `PulseEvent` missed signal issues.

Program 10–7 `QueueObjCancel:` Queue Functions Modified for Cancellation

```
/* Chapter 10. QueueObjCancel. Prepare to be cancelled */
/* Queue functions. Signal model */

#include "Everything.h"
#include "SynchObj.h"

static BOOL shutDownGet = FALSE;
static BOOL shutDownPut = FALSE;

void WINAPI QueueShutDown (DWORD n)
{
    _tprintf (_T("In ShutDownQueue callback. %d\n"), n);
    if (n%2 != 0) shutDownGet = TRUE;
    if ( (n/2) % 2 != 0) shutDownPut = TRUE;
    /* Free any resource (none in this example). */
    return;
}
```

```

DWORD QueueGet (QUEUE_OBJECT *q, PVOID msg, DWORD mSize, DWORD maxWait)
{
    if (q->msgArray == NULL) return 1; /* Queue has been destroyed */

    WaitForSingleObject (q->qGuard, INFINITE);
    while (!shutDownGet && QueueEmpty (q)) {
        if (SignalObjectAndWait (q->qGuard, q->qNe, INFINITE, TRUE) ==
            WAIT_IO_COMPLETION && shutDownGet) {
            continue;
        }
        WaitForSingleObject (q->qGuard, INFINITE);
    }
    /* remove the message from the queue */
    if (!shutDownGet) {
        QueueRemove (q, msg, mSize);
        /* Signal queue not full as we've removed a message */
        SetEvent (q->qNf);
        ReleaseMutex (q->qGuard);
    }
    return shutDownGet ? WAIT_TIMEOUT : 0;
}

DWORD QueuePut (QUEUE_OBJECT *q, PVOID msg, DWORD mSize, DWORD maxWait)
{
    if (q->msgArray == NULL) return 1; /* Queue has been destroyed */

    WaitForSingleObject (q->qGuard, INFINITE);
    while (!shutDownPut && QueueFull (q)) {
        if (SignalObjectAndWait(q->qGuard, q->qNf, INFINITE, TRUE) ==
            WAIT_IO_COMPLETION && shutDownPut) {
            continue;
        }
        WaitForSingleObject (q->qGuard, INFINITE);
    }
    /* Put the message in the queue */
    if (!shutDownPut) {
        QueueInsert (q, msg, mSize);
        /* Signal queue not empty as we've inserted a message */
        SetEvent (q->qNe);
        ReleaseMutex (q->qGuard);
    }
    return shutDownPut ? WAIT_TIMEOUT : 0;
}

```

The APC routine could be either `ShutDownReceiver` or `ShutDownTransmitter`, as the receiver and transmitter threads use both `QueueGet` and `QueuePut`. If it were necessary for the shutdown functions to know which thread

they are executed from, use different APC argument values for the third `QueueUserAPC` arguments in the code segment preceding Program 10–7.

The thread exit code will be `WAIT_TIMEOUT` to maintain consistency with previous versions. A `DllMain` function can perform additional cleanup in a `DllMain` function if appropriate.

An alternative to testing the return value for `WAIT_IO_COMPLETION` would be for the shutdown functions to raise an exception, place the `QueuePut` body in a try block, and add an exception handler.

APCs and Missed Signals

A kernel mode APC (used in asynchronous I/O) can momentarily move a waiting thread out of its wait state, potentially causing a missed `PulseEvent` signal. Some documentation warns against `PulseEvent` for this reason, as discussed earlier in the “`PulseEvent`: Another Caution” section. Should there be a situation where a missed signal could occur, include a finite time-out period on the appropriate wait calls, or use Windows NT6 condition variables. Better yet, avoid `PulseEvent`.

Safe Thread Cancellation

The preceding example and discussion show how we can safely cancel a target thread that uses alertable wait states. Such cancellation is sometimes called *synchronous cancellation*, despite the use of APCs, because the cancellation, which is caused by the boss’s `QueueUserAPC` call, can only take effect when the target thread permits cancellation by entering a safe alertable wait state.

Synchronous cancellation requires the target thread to cooperate and allow itself to be canceled from time to time. Event waits are a natural place to enter an alertable wait state because, as a system shuts down, the event may never be signaled again. Mutex waits could also be alertable to allow thread waiting on a resource that may not become available again. For example, a boss thread could break deadlocks with this technique.

Asynchronous thread cancellation might appear useful to signal a compute-bound thread that seldom, if ever, waits for I/O or events. Windows does not allow asynchronous cancellation, and it would be a risky operation. You do not know the state of the thread to be canceled and whether it owns locks or other resources. There are techniques, using processor-specific code, to interrupt a specified thread, but the techniques not only are risky but are nonportable.

Pthreads for Application Portability

Pthreads have been mentioned several times as the alternative threading and synchronization model available with UNIX, Linux, and other non-Windows systems. There is an open source Windows Pthreads library, and with this library, you can write portable threaded applications that can run on a wide variety of systems. The *Examples* file discusses this subject in more detail. The `ThreeStagePthreads` project uses the open source library and points to the download site.

Thread Stacks and the Number of Threads

Two more cautions, which are related, are in order. First, give some thought to the thread stack size, where 1MB is the default. This should be sufficient in most cases, but if there is any doubt, determine the maximum amount of stack space each thread will require, including the requirements of any library functions or recursive functions that the thread calls. A stack overflow will corrupt other memory or cause an exception.

Second, a large number of threads with large stacks will require large amounts of virtual memory for the process and could affect paging behavior and the paging file. For example, using 1,000 threads would not be unreasonable in some of the examples in this and later chapters. Allowing 1MB per thread stack results in 1GB of virtual address space. Preventive measures include careful stack sizing, thread pools, and multiplexing operations within a single thread. Furthermore, parallelism frameworks (previous chapter) generally assure that there are bounds on the total stack size and task-switching times.

Hints for Designing, Debugging, and Testing

At the risk of presenting advice that is contrary to that given in many other books and technical articles, which stress testing and little else, my personal advice is to balance your efforts so that you pay attention to design, implementation, and use of familiar programming models. The best debugging technique is not to create the bugs in the first place; this advice, of course, is easier to give than to follow. Nonetheless, when defects do occur, as they will, code inspection, balanced with debugging, often is most effective in finding and fixing the defects' root causes.

Overdependence on testing is not advisable because many serious defects will elude the most extensive and expensive testing. Testing can only reveal defects; it cannot prove that they do not exist, and testing shows only defect symptoms, not root causes. As a personal example, I ran a version of a multiple semaphore wait function that used the CV model without the finite time-out on the event variable