involves producers, consumers, and message passing; Programs 10–1 and 10–2 provide a different example.

The Condition Variable Model

Now let's combine all of this into a single code fragment that represents what we will call the *condition variable model* (CV model) with two variations, the *signal* and *broadcast* CV models. The first examples use the broadcast variation. The result is a program model that will occur frequently and can solve a wide variety of synchronization problems. For convenience, the example is stated in terms of a producer and a consumer.

The discussion may seem a bit abstract, but once the techniques are understood, we will be able to solve synchronization problems that would be very difficult without a good model.

The code fragment has several key elements.

- A data structure of type `STATE_TYPE` that contains all the data or *state variables* such as the messages, checksums, and counters used in Program 8–2.
- A mutex (alternatively, an SRW or `CRITICAL_SECTION`) and one or more events associated with, and usually a part of, the data structure.
- One or more Boolean functions to evaluate the *condition variable predicates*, which are the conditions (states) on which a thread might wait. Examples include "a new message is ready," "there is available space in the buffer," and "the queue is not empty." A distinct event may be associated with each condition variable predicate, or one event may be used to represent simply a change of state or a combination (logical "or") of several predicates. In the latter case, test individual predicate functions with the mutex locked to determine the actual state. If the predicate (logical expression) is simple, there is no need for a separate function.

The following code segment shows a producer and consumer using these principles, with a single event and condition variable predicate (implemented with a function, `cvp`, that is assumed but not shown). When the producer signals that a desired state has been reached, the signal should be broadcast to all waiting consumers. For instance, the producer may have created several messages, and the state is changed by increasing the message count. In many situations, you want to release only a single thread, as discussed after the code fragment.

This code segment is designed to operate under all NT kernel versions and even Windows 9x. `SignalObjectAndWait` will then simplify the solution. We show this full solution, appropriate for obsolete Windows versions, because:

- The usage pattern is still common in existing programs.

- Understanding the segment will make it easier to see the usefulness of `SignalObjectAndWait` and NT6 condition variables.
- The usage pattern is still useful when using a `CRITICAL_SECTION` in place of a mutex.

NT6 condition variables will further simplify and improve the solution.

Note and caution: This example deliberately uses `PulseEvent`, even though many writers and some of the Microsoft documentation warn against its use (see the remarks section in the MSDN entry). The ensuing discussion and examples will justify this choice, but with an additional cautionary note in the `SignalObjectAndWait` section. Also, there is a `WaitForSingleObject` call with a finite time-out, making the loop a form of polling loop; we show later how to eliminate the time-out.

```
typedef struct STATE_TYPE_T {
   HANDLE sGuard; /* Mutex to protect the object. */
   HANDLE cvpSet; /* Manual-reset event -- cvp () holds. */
   ... other events ...
   /* State structure with counts, checksums, etc. */
   struct STATE_VAR_TYPE stateVar;
} STATE_TYPE state;
...
/* Initialize state, creating the mutex and event. */
...
/* PRODUCER thread that modifies state. */
WaitForSingleObject (state.sGuard, INFINITE);
/* Change state so that the CV predicate holds. */
/* Example: one or more messages are now ready. */
state.stateVar.msgCount += N;
PulseEvent (state.cvpSet);
ReleaseMutex (state.sGuard);
/* End of the interesting part of the producer. */
...
/* CONSUMER thread function waits for a particular state. */
WaitForSingleObject (state.sGuard, INFINITE);
while (!cvp (&state)) {
   ReleaseMutex (state.sGuard);
   /* The timeout is required, making this loop poll */
   WaitForSingleObject (state.cvpSet, TimeOut);
   WaitForSingleObject (state.sGuard, INFINITE);
}
```

```
/* This thread now owns the mutex and cvp (&state) holds. */
/* Take appropriate action, perhaps modifying state. */
...
   ReleaseMutex (state.sGuard);
/* End of the interesting part of the consumer. */
```

Comments on the Condition Variable Model

The essential feature in the code segment is the loop in the consumer code. The loop body consists of three steps: (1) unlock the mutex that was locked prior to entering the loop; (2) wait, with a finite time-out, on the event; and (3) lock the mutex again. *The event wait time-out is significant,* as explained later.

Pthreads, as implemented in many UNIX and other systems, combine these three steps into a single function, `pthread_cond_wait`, combining a mutex and a condition variable (which is similar but not identical to the Windows event). Windows NT6 condition variables do the same thing. This is the reason for the term "condition variable model." There is also a timed version, which allows a time-out on the event wait.

Importantly, the single Pthreads function implements the first two steps (the mutex release and event wait) as an atomic operation so that no other thread can run before the calling thread waits on the event (or condition variable).

The Pthreads designers and the NT6 designers made a wise choice; the two functions (with and without a time-out) are the only ways to wait on a condition variable in Pthreads, so a condition variable must always be used with a mutex. Windows (before NT6) forces you to use two or three separate function calls, and you need to do it in just the right way to avoid problems.

Another motivation for learning the CV model, besides simplifying programs, is that it is essential if you ever need to use Pthreads or convert a Pthreads program to Windows.

Note: Windows NT Version 4.0 introduced a new function, `SignalObject-AndWait` (SOAW), that performs the first two steps atomically. The later examples assume that this function is available, in keeping with the policy established in Chapter 1. Nonetheless, the CV model introduction does not use SOAW in order to motivate its later usage, and a few examples have alternative implementations on the book's *Examples* file that use a CS in place of a mutex. (SOAW cannot be used with a CS.) Appendix C (Table C–6) shows that `SignalObjectAndWait` provides significant performance advantages.

Using the Condition Variable Model

The CV model, when implemented properly, works as follows in the producer/consumer context.

- The producer locks the mutex, changes state, pulses the event when appropriate, and unlocks the mutex. For example, the producer pulses the event when one or more messages are ready.
- The `PulseEvent` call should be with the mutex locked so that no other thread can modify the object, perhaps invalidating the condition variable predicate.
- The consumer tests the condition variable predicate *with the mutex locked.* If the predicate holds, there is no need to wait.
- If the predicate does not hold, the consumer must unlock the mutex before waiting on the event. Otherwise, no thread could ever modify the state and set the event.
- The event wait must have a time-out just in case the producer pulses the event in the interval between the mutex release (step 1) and the event wait (step 2). That is, without the *finite* time-out, there could be a "lost signal," which is another example of a race condition. APCs, described later in this chapter, can also cause lost signals. The time-out value used in the producer/consumer segment is a tunable parameter. (See Appendix C for comments on optimal values.)
- The consumer always retests the predicate after the event wait. Among other things, this is necessary in case the event wait has timed out. Also, the state may have changed. For example, the producer may have produced two messages and then released three waiting consumers, so one of the consumers will test the state, find no more messages, and wait again. Finally, the retest protects against spurious wakeups that might result from a thread setting or pulsing the event without the mutex locked. There is no way to avoid this time-out and polling loop until we get to `SignalObjectAndWait` and the Windows NT6 condition variables.
- The consumer always owns the mutex when it leaves the loop, regardless of whether the loop body was executed.

Condition Variable Model Variations

Notice, first, that the preceding code fragment uses a manual-reset event and calls `PulseEvent` rather than `SetEvent`. Is this the correct choice, and could the event be used differently? The answer is yes to both questions.

Referring back to Table 8–1, we see that the example has the property that *multiple threads* will be released. This is correct in this example, where several messages are produced and there are multiple consuming threads, and we need to broadcast the change. However, if the producer creates just one message and there are multiple consuming threads, the event should be auto-reset and the pro-

ducer should call `SetEvent` to ensure that exactly one thread is released. This variation is the "signal CV" model rather than the "broadcast CV" model. It is still essential for the released consumer thread, which will then own the mutex and can remove a message.

Of the four combinations in Table 8–1, two are useful in the CV model. Considering the other two combinations, auto-reset/`PulseEvent` would have the same effect as auto-reset/`SetEvent` (the signal CV model) because of the time-out, but the dependence on the time-out would reduce responsiveness. The manual-reset/`SetEvent` combination causes spurious signals (the condition variable predicate test offers protection, however), because some thread must reset the event, and there will be a race among the threads before the event is reset.

In summary:

- Auto-reset/`SetEvent` is the signal CV model, which releases a single waiting thread.
- Manual-reset/`PulseEvent` is the broadcast CV model, which releases all waiting threads.
- Pthreads and NT6 condition variables make the same distinction but do not require the finite time-out in the event wait for the broadcast model, whereas the time-out is essential in Windows because the mutex release and event wait are not performed atomically.
- This will change, however, when we introduce `SignalObjectAndWait`.

An Example Condition Variable Predicate

Consider the condition variable predicate:

```
state.stateVar.count >= K;
```

In this case, a consumer thread will wait until the count is sufficiently large. This shows, for example, how to implement a multiple-wait semaphore; recall that normal semaphores do not have an atomic wait for multiple units. The consumer thread would then decrement the count by `K` after leaving the loop but before releasing the mutex.

Notice that the broadcast CV model is appropriate in this case because a single producer may increase the count so as to satisfy several but not all of the waiting consumers.

Semaphores and the Condition Variable Model

In some cases, a semaphore would be appropriate rather than an event, and semaphores have the advantage of specifying the exact number of threads to be released. For example, if each consumer were known to consume exactly one message, the producer could call `ReleaseSemaphore` with the exact number of messages produced. In the more general case, however, the producer does not know how the individual consumers will modify the state variable structure, so the CV model can solve a wider class of problems.

The CV model is powerful enough to implement semaphores. As described earlier, the basic technique is to define a predicate stating that "the semaphore count is nonzero" and create a state structure containing the count and maximum value. Exercise 10–10 shows a complete solution that allows for an atomic wait for multiple units.

Using `SignalObjectAndWait`

The consumer loop in the preceding code segment is critical to the CV model because it waits for a state change and then tests to see if the desired state holds. The state may not hold if the event is too *coarse,* indicating, for example, that there was simply some state change, not necessarily the required change. Furthermore, a different consumer thread might have made some other state change, such as emptying the message buffer. The loop required two waits and a mutex release, as follows:

```
while (!cvp (&state)) {
   ReleaseMutex (state.sGuard);
   WaitForSingleObject (state.cvpSet, TimeOut);
   WaitForSingleObject (state.sGuard, INFINITE);
}
```

The time-out on the first wait (the event wait) is necessary in order to avoid missed signals and other potential problems. This code will work if you replace the mutexes with CSs.

SOAW is an important enhancement that eliminates the need for the time-out and combines the first two loop statements; that is, the mutex release and the event wait. In addition to the program simplicity benefit, performance generally improves because a system call is eliminated and there is no need to tune the wait time-out period.

```
DWORD SignalObjectAndWait (
   HANDLE hObjectToSignal,
   HANDLE hObjectToWaitOn,
   DWORD dwMilliseconds,
   BOOL bAlertable)
```

This function simplifies the consumer loop, where the two handles are the mutex and event handles, respectively. There is no event wait time-out because the calling thread waits on the second handle *immediately* after the first handle is signaled (which, in this case, means that the mutex is released). The signal and wait are atomic so that no other thread can possibly signal the event between the time that the calling thread releases the mutex and the thread waits on the second handle. The simplified consumer loop, then, is as follows.

```
while (!cvp (&state)) {
   SignalObjectAndWait (state.sGuard, state.cvpSet,
      INFINITE, FALSE);
   WaitForSingleObject (state.sGuard, INFINITE);
}
```

The final argument, `bAlertable`, is `FALSE` here but will be set to `TRUE` in the later sections on APCs.

In general, the two handles can be for any appropriate synchronization objects. You cannot, however, use a `CRITICAL_SECTION` as the signaled object; kernel objects are necessary.

Many program examples, both in the book and in the *Examples* file, use `SignalObjectAndWait`, although some alternative solutions are also included and are mentioned in the text. If you want to use a `CRITICAL_SECTION` instead of a mutex, use the signal/wait pair in the original code segment and be certain to have a finite time-out period on the event wait.

The section on APCs shows a different technique to signal waiting threads with the additional advantage of signaling a specific waiting thread, whereas, when using events, there is no easy way to control which thread is signaled.

`PulseEvent`: One More Caution

`SignalObjectAndWait`, used with `PulseEvent`, appears to implement the broadcast CV model properly, and it nearly does. The remaining problem is that the Windows executive can, under some circumstances, preempt a waiting thread

(such as one waiting on `SignalObjectAndWait`) just as another thread calls `PulseEvent`, resulting in a missed signal and possibly a permanently blocked thread. This is `PulseEvent`'s fatal flaw and the principal reason that MSDN warns against it.

Unfortunately, it's frequently necessary to use the CV model (e.g., you need to port a Pthreads program or you need the underlying functionality). Fortunately, there are several defenses against this flaw:

- Use a finite time-out with SOAW, treating a time-out as a spurious signal detected when the loop retests the predicate. You could also gain performance using a CS or an SRW lock rather than a mutex and then wait on the event with a finite time-out.
- Use the much faster Windows condition variables if you do not need to support NT5 (Windows XP, etc.).
- Assure that your program never uses the functions that would cause the executive to preempt a waiting thread. `GetThreadContext` is one such function and is very rare. However, if you are writing a library, there is no direct way to assure that the calling program respects such limitations.
- Pulse the event multiple times so that the waiting thread will eventually receive the signal. This is the approach used in Programs 10–3 and 10–4 where new messages are generated continuously.
- Avoid the broadcast model, which we can do in Programs 10–4 and 10–5. The signal model is sufficient if you need to signal only a single thread.
- Program 10–4 uses `PulseEvent`, but comments after the program describe variations that do not require it.

Example: A Threshold Barrier Object

Suppose that you wish to have the worker threads wait until there are enough workers to form a work crew to work in parallel on a task, as in Program 7–1 (`sortMT`). Or, you may want to wait until all threads have finished the first phase of a parallel computation before proceeding to the next phase. Once the threshold is reached, all the workers start operation, and if any other workers arrive later, they do not wait. This problem is solvable with a threshold barrier compound object.

Programs 10–1 and 10–2 show the implementation of the three functions that support the threshold barrier compound object. Two of the functions, `CreateThresholdBarrier` and `CloseThresholdBarrier`, manage a `THB_OBJECT`. The threshold number of threads is a parameter to `CreateThresholdBarrier`.

Program 10–1 shows the appropriate part of the header file, `SynchObj.h`, while Program 10–2 shows the implementation of the three functions. Notice that the barrier object has a mutex, an event, a counter, and a threshold. The condition variable predicate is documented in the header file—that is, the event is to be set exactly when the count is greater than or equal to the threshold.

Program 10–1 `SynchObj.h`: Part 1—Threshold Barrier Definitions

```
/* THRESHOLD BARRIER - TYPE DEFINITION AND FUNCTIONS */
typedef struct THRESHOLD_BARRIER_TAG { /* Threshold Barrier */
   HANDLE bGuard;/* mutex for the object */
   HANDLE bEvent;/* auto-reset event: bCount >= bThreshold */
   DWORD bCount; /* number of threads that have reached the Barrier */
   DWORD bThreshold;/* Barrier threshold */
} THRESHOLD_BARRIER, *THB_OBJECT;

DWORD CreateThresholdBarrier (THB_OBJECT *, DWORD /* threshold */);
DWORD WaitThresholdBarrier (THB_OBJECT);
DWORD CloseThresholdBarrier (THB_OBJECT);

/* Error Values */
#define SYNCH_OBJ_NOMEM 1  /* Unable to allocate resources */
#define SYNCH_OBJ_CREATE_FAILURE 2
#define SYNCH_OBJ_BUSY 3  /* Object is in use and cannot be closed */
```

Program 10–2 now shows the implementation of the three functions. A test program, `testTHB.c`, is in the *Examples* file. Notice how the `WaitThreshold-Barrier` function contains the familiar condition variable loop. Also notice that the wait function not only waits on the event but also signals the event. The previous producer/consumer example waited and signaled in separate functions.

Finally, the condition variable predicate is, in this case, persistent. Once it becomes true, it will never change, unlike the situation in other examples. This allows a further simplification in `WaitThresholdBarrier`. `SetEvent` is okay because there is no need to reset the event, although `PulseEvent` would also work and would adhere to the CV model. Later examples do use the CV model.

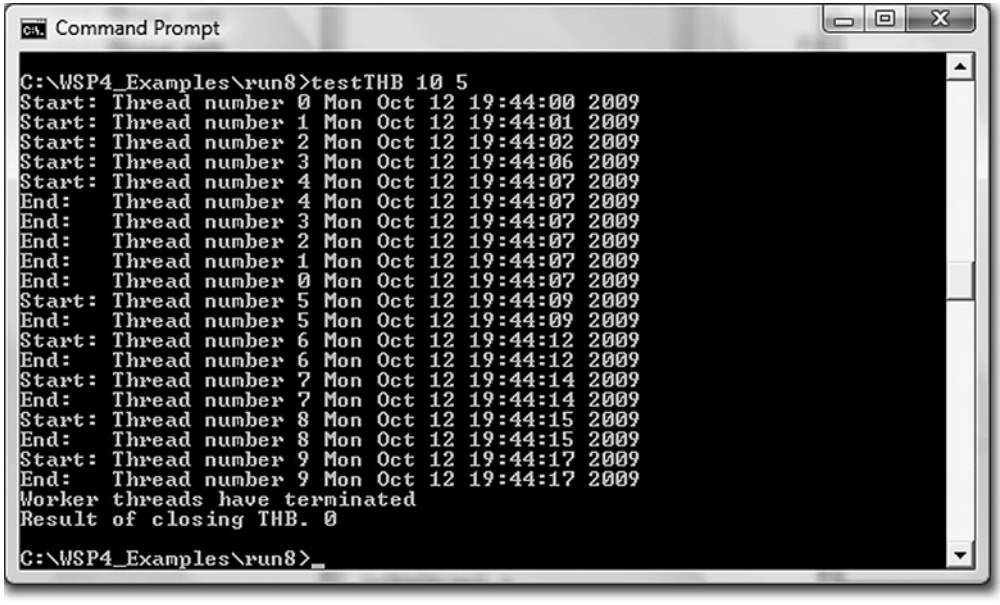
Program 10–2 `ThbObject`: Implementing the Threshold Barrier

```
/* ThbObject. Program 10-2          */
/* threshold Barrier compound synch objects library*/

#include "Everything.h"
#include "synchobj.h"
```

```
/**********************/
/*  THRESHOLD barrier OBJECTS */
/**********************/

DWORD CreateThresholdBarrier (THB_OBJECT *pThb, DWORD bValue)
{
    THB_OBJECT objThb;
    /* Initialize a barrier object */
    objThb = malloc (sizeof(THRESHOLD_BARRIER));
    if (objThb == NULL) return SYNCH_OBJ_NOMEM;

    objThb->bGuard = CreateMutex (NULL, FALSE, NULL);
    if (objThb->bGuard == NULL) return SYNCH_OBJ_CREATE_FAILURE;

    /* Manual reset event */
    objThb->bEvent = CreateEvent (NULL, TRUE, FALSE, NULL);
    if (objThb->bEvent == NULL) return SYNCH_OBJ_CREATE_FAILURE;

    objThb->bThreshold = bValue;
    objThb->bCount = 0;

    *pThb = objThb;

    return 0;
}

DWORD WaitThresholdBarrier (THB_OBJECT thb)
{
    /* Wait for the specified number of threads to reach */
    /* the barrier, then broadcast on the CV */

    WaitForSingleObject (thb->bGuard, INFINITE);
    thb->bCount++;  /* A new thread has arrived */
    while (thb->bCount < thb->bThreshold) {
        SignalObjectAndWait(thb->bGuard, thb->bEvent, INFINITE, FALSE);
        WaitForSingleObject(thb->bGuard, INFINITE);
    }
    SetEvent (thb->bEvent); /* Broadcast to all waiting threads */
    /* NOTE: We are broadcasting to all waiting threads.
     * HOWEVER, SetEvent is OK because the condition is persistent
     * and there is no need to reset the event. */
    ReleaseMutex (thb->bGuard);
    return 0;
}

DWORD CloseThresholdBarrier (THB_OBJECT thb)
{
    /* Destroy component mutex and event once it is safe to do so */
    WaitForSingleObject (thb->bGuard, INFINITE);
```

```
    /* Be certain that no thread is waiting on the object. */
    if (thb->bCount < thb->bThreshold) {
        ReleaseMutex (thb->bGuard);
        return SYNCH_OBJ_BUSY;
    }

    /* Now release the mutex and close the handle */
    ReleaseMutex (thb->bGuard);
    CloseHandle (thb->bEvent);
    CloseHandle (thb->bGuard);
    free (thb);
    return 0;
}
```

Run 10–2 shows the test program, `testTHB`, with command line parameters to start 10 short-lived threads and a barrier of 5. Each thread prints its start and stop time, and `testTHB` starts new threads at random intervals averaging one thread every 1.5 seconds (approximately). The first five threads end at about the same time immediately after the fifth thread arrives. Later threads end shortly after they start.



Run 10–2 `testTHB`**:** Testing the Threshold Barrier Functions

Comments on the Threshold Barrier Implementation

The threshold barrier object implemented here is limited for simplicity. In general, we would want to emulate Windows objects more closely by:

- Allowing the object to have security attributes (Chapter 15)
- Allowing the object to be named
- Permitting multiple objects on the object and not destroying it until the reference count is 0
- Allowing the object to be shared between processes

The *Examples* file contains a full implementation of one such object, a multiple wait semaphore, and the techniques used there can then be used for any of the objects in this chapter.

A Queue Object

So far, we have associated a single event with each mutex, but in general there might be more than one condition variable predicate. For example, in implementing a first in, first out (FIFO) queue, a thread that removes an element from the queue needs to wait on an event signifying that the queue is not empty, while a thread placing an element in the queue must wait until the queue is not full. The solution is to provide two events, one for each condition. Notice, however, that there is a single mutex.

Program 10–3 shows the definitions of a queue object and its functions. Programs 10–4 and 10–5 show the queue functions and a program that uses them.

Program 10–3 `SynchObj.h`: Part 2—Queue Definitions

```
/* Definitions of synchronized, general bounded queue structure. */
/* Queues are implemented as arrays with indices to youngest */
/* and oldest messages, with wrap around. */
/* Each queue also contains a guard mutex and */
/* "not empty" and "not full" condition variables. */
/* Finally, there is a pointer to an array of character messages. */

typedef struct QUEUE_OBJECT_TAG { /* General-purpose queue */
    HANDLE qGuard;/* Guard the message block*/
    HANDLE qNe;/* Event: Queue is not empty*/
    HANDLE qNf;/* Event: Queue is not full*/
           /* These two events are manual-reset for the broadcast model
            * and auto-reset for the signal model */
    DWORD qSize;/* Queue max size size*/
```