

header file. Document it in the project design notes. Write it on your whiteboard. And then tie a string around your finger to be sure that you do not forget.

You are free to unlock the mutexes in whatever order makes the most sense. Unlocking mutexes cannot result in deadlock. In the next section, I will talk about a sort of "overlapping hierarchy" of mutexes, called a "lock chain," where the normal mode of operation is to lock one mutex, lock the next, unlock the first, and so on. If you use a "try and back off" algorithm, however, you should always try to release the mutexes in reverse order. That is, if you lock mutex 1, mutex 2, and then mutex 3, you should unlock mutex 3, then mutex 2, and finally mutex 1. If you unlock mutex 1 and mutex 2 while mutex 3 is still locked, another thread may have to lock both mutex 1 and mutex 2 before finding it cannot lock the entire hierarchy, at which point it will have to unlock mutex 2 and mutex 1, and then retry. Unlocking in reverse order reduces the chance that another thread will need to back off.

### 3.2.5.2 Lock chaining

"Chaining" is a special case of locking hierarchy, where the scope of two locks overlap. With one mutex locked, the code enters a region where another mutex is required. After successfully locking that second mutex, the first is no longer needed, and can be released. This technique can be very valuable in traversing data structures such as trees or linked lists. Instead of locking the entire data structure with a single mutex, and thereby preventing any parallel access, each node or link has a unique mutex. The traversal code would first lock the queue head, or tree root, find the desired node, lock it, and then release the root or queue head mutex.

Because chaining is a special form of hierarchy, the two techniques are compatible, if you apply them carefully. You might use hierarchical locking when balancing or pruning a tree, for example, and chaining when searching for a specific node.

Apply lock chaining with caution, however. It is exceptionally easy to write code that spends most of its time locking and unlocking mutexes that never exhibit any contention, and that is wasted processor time. Use lock chaining only when multiple threads will almost always be active within different parts of the hierarchy.

## 3.3 Condition variables

*"There's no sort of use in knocking," said the Footman, "and that for two reasons. First, because I'm on the same side of the door as you are: secondly, because they're making such a noise inside, no one could possibly hear you."*

*—Lewis Carroll, Alice's Adventures in Wonderland*

Butenhot, Programming with POSIX Threads Addison-Wesley

Co:

FIG

sh:  
lon  
or

cor  
nal  
rov  
ber  
"w:  
int  
ing

fin  
Th  
the  
ser  
an

mu  
del

whiteboard.  
 forget.  
 most sense.  
 n, I will talk  
 chain," where  
 lock the first,  
 should always  
 x 1, mutex 2,  
 ally mutex 1.  
 other thread  
 not lock the  
 mutex 1, and  
 er thread will

of two locks  
 her mutex is  
 is no longer  
 in traversing  
 e entire data  
 access, each  
 k the queue  
 e the root or

ies are com-  
 ocking when  
 ng for a spe-

asy to write  
 s that never  
 haining only  
 parts of the



**FIGURE 3.3** Condition variable analogy

A condition variable is used for communicating information about the state of shared data. You would use a condition variable to signal that a queue was no longer empty, or that it had become empty, or that anything else needs to be done or can be done within the shared data manipulated by threads in your program.

Our seafaring programmers use a mechanism much like condition variables to communicate (Figure 3.3). When the rower nudges a sleeping programmer to signal that the sleeping programmer should wake up and start rowing, the original rower "signals a condition." When the exhausted ex-rower sinks into a deep slumber, secure that another programmer will wake him at the appropriate time, he is "waiting on a condition." When the horrified bailer discovers that water is seeping into the boat faster than he can remove it, and he yells for help, he is "broadcasting a condition."

When a thread has mutually exclusive access to some shared state, it may find that there is no more it can do until some other thread changes the state. The state may be correct, and consistent—that is, no invariants are broken—but the current state just doesn't happen to be of interest to the thread. If a thread servicing a queue finds the queue empty, for example, the thread must wait until an entry is added to the queue.

The shared data, for example, the queue, is protected by a mutex. A thread must lock the mutex to determine the current state of the queue, for example, to determine that it is empty. The thread must unlock the mutex before waiting (or

no other thread would be able to insert an entry onto the queue), and then it must wait for the state to change. The thread might, for example, by some means block itself so that a thread inserting a new queue entry can find its identifier and awaken it. There is a problem here, though—the thread is running between unlocking and blocking.

If the thread is still running while another thread locks the mutex and inserts an entry onto the queue, that other thread cannot determine that a thread is waiting for the new entry. The waiting thread has already looked at the queue and found it empty, and has unlocked the mutex, so it will now block itself without knowing that the queue is no longer empty. Worse, it may not yet have recorded the fact that it intends to wait, so it may wait forever because the other thread cannot find its identifier. The unlock and wait operations must be atomic, so that no other thread can lock the mutex before the waiter has become blocked, and is in a state where another thread can awaken it.

! A condition variable wait always returns with the mutex locked.

That's why *condition variables* exist. A condition variable is a “signaling mechanism” associated with a mutex and by extension is also associated with the shared data protected by the mutex. *Waiting* on a condition variable atomically releases the associated mutex and waits until another thread *signals* (to wake one waiter) or *broadcasts* (to wake all waiters) the condition variable. The mutex must always be locked when you wait on a condition variable and, when a thread wakes up from a condition variable wait, it always resumes with the mutex locked.

The shared data associated with a condition variable, for example, the queue “full” and “empty” conditions, are the *predicates* we talked about in Section 3.1. A condition variable is the mechanism your program uses to wait for a predicate to become true, and to communicate to other threads that it might be true. In other words, a condition variable allows threads using the queue to exchange information about the changes to the queue state.

! Condition variables are for *signaling*, not for mutual exclusion.

Condition variables do not provide mutual exclusion. You need a mutex to synchronize access to the shared data, including the predicate for which you wait. That is why you must specify a mutex when you wait on a condition variable. By making the unlock atomic with the wait, the Pthreads system ensures that no thread can change the predicate after you have unlocked the mutex but before your thread is waiting on the condition variable.

Why isn't the mutex created as part of the condition variable? First, mutexes are used separately from any condition variable as often as they're used with condition variables. Second, it is common for one mutex to have more than one associated condition variable. For example, a queue may be “full” or “empty.” Although you may have two condition variables to allow threads to wait for either

coi  
qu

sh:  
va  
no  
fus  
ris  
yo  
br

gra  
yo  
is  
po  
va

an  
co

condition, you must have one and only one mutex to synchronize *all* access to the queue header.

A condition variable should be associated with a single predicate. If you try to share one condition variable between several predicates, or use several condition variables for a single predicate, you're risking deadlock or race problems. There's nothing wrong with doing either, as long as you're careful—but it is easy to confuse your program (computers aren't very smart) and it is usually not worth the risk. I will expound on the details later, but the rules are as follows: First, when you share a condition variable between multiple predicates, you must always *broadcast*, never *signal*; and second, *signal* is more efficient than *broadcast*.

Both the condition variable and the predicate are shared data in your program; they are used by multiple threads, possibly at the same time. Because you're thinking of the condition variable and predicate as being locked together, it is easy to remember that they're always controlled using the same mutex. It is possible (and legal, and often even reasonable) to *signal* or *broadcast* a condition variable without having the mutex locked, but it is safer to have it locked.

Figure 3.4 is a timing diagram showing how three threads, thread 1, thread 2, and thread 3, interact with a condition variable. The rounded box represents the condition variable, and the three lines represent the actions of the three threads.

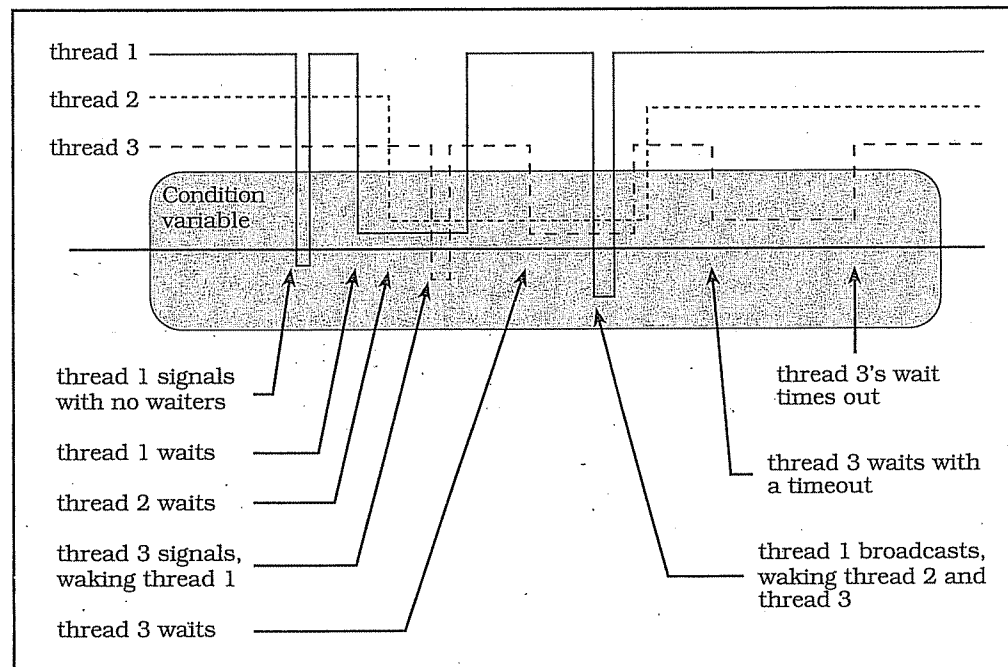


FIGURE 3.4 Condition variable operation

, and then it  
some means  
identifier and  
ning between

x and inserts  
t a thread is  
he queue and  
itself without  
ave recorded  
other thread  
omic, so that  
ocked, and is

nalng mech-  
ted with the  
le atomically  
uals (to wake  
e. The mutex  
hen a thread  
h the mutex

le, the queue  
ection 3.1. A  
predicate to  
ue. In other  
nge informa-

a mutex to  
r which you  
ndition vari-  
tem ensures  
e mutex but

rst, mutexes  
ed with con-  
re than one  
or "empty."  
ait for either

When a line goes within the box, it is “doing something” with the condition variable. When a thread’s line stops before reaching below the middle line through the box, it is waiting on the condition variable; and when a thread’s line reaches below the middle line, it is signaling or broadcasting to awaken waiters.

Thread 1 signals the condition variable, which has no effect since there are no waiters. Thread 1 then waits on the condition variable. Thread 2 also blocks on the condition variable and, shortly thereafter, thread 3 signals the condition variable. Thread 3’s signal unblocks thread 1. Thread 3 then waits on the condition variable. Thread 1 broadcasts the condition variable, unblocking both thread 2 and thread 3. Thread 3 waits on the condition variable shortly thereafter, with a timed wait. Some time later, thread 3’s wait times out, and the thread awakens.

### 3.3.1 Creating and destroying a condition variable

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init (pthread_cond_t *cond,
    pthread_condattr_t *condattr);
int pthread_cond_destroy (pthread_cond_t *cond);
```

A condition variable is represented in your program by a variable of type `pthread_cond_t`. You should never make a copy of a condition variable, because the result of using a copied condition variable is undefined. It would be like telephoning a disconnected number and expecting an answer. One thread could, for example, wait on one copy of the condition variable, while another thread signaled or broadcast the other copy of the condition variable—the waiting thread would not be awakened. You can, however, freely pass pointers to a condition variable so that various functions and threads can use it for synchronization.

Most of the time you’ll probably declare condition variables using the `extern` or `static` storage class at file scope, that is, outside of any function. They should have normal (`extern`) storage class if they are used by other files, or `static` storage class if used only within the file that declares the variable. When you declare a static condition variable that has default attributes, you should use the `PTHREAD_COND_INITIALIZER` initialization macro, as shown in the following example, `cond_static.c`.

■ `cond_static.c`

```
1 #include <pthread.h>
2 #include "errors.h"
3
4 /*
5  * Declare a structure, with a mutex and condition variable,
6  * statically initialized. This is the same as using
```

C

```
7
8
9
10 t
11
12
13
14 }
15
16 m
17
18
19 i
20 {
21
22 }
```

■

a  
s  
c  
h  
n  
v  
t  
c  
c  
r  
e  
e  
t  
s

■

```
1 #
2 #
3
4 /
5
6
7 t
```

condition variable through line reaches

rs. there are no so blocks on condition variable the condition both thread 2 after, with a thread 1 awakens.

able of type variable, because be like tele- id could, for thread sig- iting thread a condition ization.

the extern They should static stor- you declare id use the oving exam-

```

7  * pthread_mutex_init and pthread_cond_init, with the default
8  * attributes.
9  */
10 typedef struct my_struct_tag {
11     pthread_mutex_t  mutex; /* Protects access to value */
12     pthread_cond_t   cond; /* Signals change to value */
13     int               value; /* Access protected by mutex */
14 } my_struct_t;
15
16 my_struct_t data = {
17     PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, 0};
18
19 int main (int argc, char *argv[])
20 {
21     return 0;
22 }

```

■ cond\_static.c

Condition variables and their predicates are “linked”—for best results, treat them that way!

When you declare a condition variable, remember that a condition variable and the associated predicate are “locked together.” You may save yourself (or your successor) some confusion by always declaring the condition variable and predicate together, if possible. I recommend that you try to encapsulate a set of invariants and predicates with its mutex and one or more condition variables as members in a structure, and carefully document the association.

Sometimes you cannot initialize a condition variable statically; for example, when you use malloc to create a structure that contains a condition variable. Then you will need to call pthread\_cond\_init to initialize the condition variable dynamically, as shown in the following example, cond\_dynamic.c. You can also dynamically initialize condition variables that you declare statically—but you must ensure that each condition variable is initialized before it is used, and that each is initialized only once. You may initialize it before creating any threads, for example, or by using pthread\_once (Section 5.1). If you need to initialize a condition variable with nondefault attributes, you must use dynamic initialization (see Section 5.2.2).

■ cond\_dynamic.c

```

1 #include <pthread.h>
2 #include "errors.h"
3
4 /*
5  * Define a structure, with a mutex and condition variable.
6  */
7 typedef struct my_struct_tag {

```

```

8   pthread_mutex_t  mutex; /* Protects access to value */
9   pthread_cond_t   cond;  /* Signals change to value */
10  int              value; /* Access protected by mutex */
11 } my_struct_t;
12
13 int main (int argc, char *argv[])
14 {
15     my_struct_t *data;
16     int status;
17
18     data = malloc (sizeof (my_struct_t));
19     if (data == NULL)
20         errno_abort ("Allocate structure");
21     status = pthread_mutex_init (&data->mutex, NULL);
22     if (status != 0)
23         err_abort (status, "Init mutex");
24     status = pthread_cond_init (&data->cond, NULL);
25     if (status != 0)
26         err_abort (status, "Init condition");
27     status = pthread_cond_destroy (&data->cond);
28     if (status != 0)
29         err_abort (status, "Destroy condition");
30     status = pthread_mutex_destroy (&data->mutex);
31     if (status != 0)
32         err_abort (status, "Destroy mutex");
33     (void)free (data);
34     return status;
35 }

```

cond\_dynamic.c

When you dynamically initialize a condition variable, you should destroy the condition variable when you no longer need it, by calling `pthread_cond_destroy`. You do not need to destroy a condition variable that was statically initialized using the `PTHREAD_COND_INITIALIZER` macro.

It is safe to destroy a condition variable when you know that no threads can be blocked on the condition variable, and no additional threads will try to wait on, signal, or broadcast the condition variable. The best way to determine this is usually within a thread that has just successfully broadcast to unblock all waiters, when program logic ensures that no threads will try to use the condition variable later.

When a thread removes a structure containing a condition variable from a list, for example, and then broadcasts to awaken any waiters, it is safe (and also a very good idea) to destroy the condition variable before freeing the storage that the condition variable occupies. The awakened threads should check their wait predicate when they resume, so you must make sure that you don't free resources required for the predicate before they've done so—this may require additional synchronization.

### 3.3.2 Waiting on a condition variable

```
int pthread_cond_wait (pthread_cond_t *cond,
pthread_mutex_t *mutex);
int pthread_cond_timedwait (pthread_cond_t *cond,
pthread_mutex_t *mutex,
struct timespec *expiration);
```

Each condition variable must be associated with a specific mutex, and with a predicate condition. When a thread waits on a condition variable it must always have the associated mutex locked. Remember that the condition variable wait operation will *unlock* the mutex for you before blocking the thread, and it will *relock* the mutex before returning to your code.

All threads that wait on any one condition variable concurrently (at the same time) must specify the *same* associated mutex. Pthreads does not allow thread 1, for example, to wait on condition variable A specifying mutex A while thread 2 waits on condition variable A specifying mutex B. It is, however, perfectly reasonable for thread 1 to wait on condition variable A specifying mutex A while thread 2 waits on condition variable B specifying mutex A. That is, each condition variable must be associated, at any given time, with only one mutex—but a mutex may have any number of condition variables associated with it.

It is important that you test the predicate after locking the appropriate mutex and before waiting on the condition variable. If a thread signals or broadcasts a condition variable while no threads are waiting, nothing happens. If some other thread calls `pthread_cond_wait` right after that, it will keep waiting regardless of the fact that the condition variable was just signaled, which means that if a thread waits when it doesn't have to, it may never wake up. Because the mutex remains locked until the thread is blocked on the condition variable, the predicate cannot become set between the predicate test and the wait—the mutex is locked and no other thread can change the shared data, including the predicate.

! Always test your predicate; and then test it again!

It is equally important that you test the predicate again when the thread wakes up. You should always wait for a condition variable in a loop, to protect against program errors, multiprocessor races, and spurious wakeups. The following short program, `cond.c`, shows how to wait on a condition variable. Proper predicate loops are also shown in all of the examples in this book that use condition variables, for example, `alarm_cond.c` in Section 3.3.4.

20-37 The `wait_thread` sleeps for a short time to allow the main thread to reach its condition wait before waking it, sets the shared predicate (`data.value`), and then signals the condition variable. The amount of time for which `wait_thread` will sleep is controlled by the hibernation variable, which defaults to one second.

\*/  
/  
: \*/

destroy the  
\_destroy.  
initialized

reads can  
to wait on,  
is usu-  
ll waiters,  
n variable

rom a list,  
nd also a  
rage that  
their wait  
don't free  
y require

51-52 If the program was run with an argument, interpret the argument as an integer value, which is stored in `hibernation`. This controls the amount of time for which `wait_thread` will sleep before signaling the condition variable.

68-83 The main thread calls `pthread_cond_timedwait` to wait for up to two seconds (from the current time). If `hibernation` has been set to a value of greater than two seconds, the condition wait will time out, returning `ETIMEDOUT`. If `hibernation` has been set to two, the main thread and `wait_thread` race, and, in principle, the result could differ each time you run the program. If `hibernation` is set to a value less than two, the condition wait should not time out.

cond.c

```

1 #include <pthread.h>
2 #include <time.h>
3 #include "errors.h"
4
5 typedef struct my_struct_tag {
6     pthread_mutex_t  mutex; /* Protects access to value */
7     pthread_cond_t   cond; /* Signals change to value */
8     int              value; /* Access protected by mutex */
9 } my_struct_t;
10
11 my_struct_t data = {
12     PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, 0};
13
14 int hibernation = 1; /* Default to 1 second */
15
16 /*
17  * Thread start routine. It will set the main thread's predicate
18  * and signal the condition variable.
19  */
20 void *
21 wait_thread (void *arg)
22 {
23     int status;
24
25     sleep (hibernation);
26     status = pthread_mutex_lock (&data.mutex);
27     if (status != 0)
28         err_abort (status, "Lock mutex");
29     data.value = 1; /* Set predicate */
30     status = pthread_cond_signal (&data.cond);
31     if (status != 0)
32         err_abort (status, "Signal condition");
33     status = pthread_mutex_unlock (&data.mutex);
34     if (status != 0)
35         err_abort (status, "Unlock mutex");
36     return NULL;
37 }

```

```

38
39 ir
40 {
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86

```

```

87     status = pthread_mutex_unlock (&data.mutex);
88     if (status != 0)
89         err_abort (status, "Unlock mutex");
90     return 0;
91 }

```

cond.c

There are a lot of reasons why it is a good idea to write code that does not assume the predicate is always true on wakeup, but here are a few of the main reasons:

**Intercepted wakeups:** Remember that threads are asynchronous. Waking up from a condition variable wait involves locking the associated mutex. But what if some other thread acquires the mutex first? It may, for example, be checking the predicate before waiting itself. It doesn't have to wait, since the predicate is now true. If the predicate is "work available," it will accept the work. When it unlocks the mutex there may be no more work. It would be expensive, and usually counterproductive, to ensure that the latest awakened thread got the work.

**Loose predicates:** For a lot of reasons it is often easy and convenient to use approximations of actual state. For example, "there may be work" instead of "there is work." It is often much easier to signal or broadcast based on "loose predicates" than on the real "tight predicates." If you always test the tight predicates before and after waiting on a condition variable, you're free to signal based on the loose approximations when that makes sense. And your code will be much more robust when a condition variable is signaled or broadcast accidentally. Use of loose predicates or accidental wakeups may turn out to be a performance issue; but in many cases it won't make a difference.

**Spurious wakeups:** This means that when you wait on a condition variable, the wait may (occasionally) return when no thread specifically broadcast or signaled that condition variable. Spurious wakeups may sound strange, but on some multiprocessor systems, making condition wakeup completely predictable might substantially slow all condition variable operations. The race conditions that cause spurious wakeups should be considered rare.

It usually takes only a few instructions to retest your predicate, and it is a good programming discipline. Continuing without retesting the predicate could lead to serious application errors that might be difficult to track down later. So don't make assumptions: Always wait for a condition variable in a while loop testing the predicate.

You can also use the pthread\_cond\_timedwait function, which causes the wait to end with an ETIMEDOUT status after a certain time is reached. The time is an absolute clock time, using the POSIX.1b struct timespec format. The timeout is absolute rather than an interval (or "delta time") so that once you've computed the timeout it remains valid regardless of spurious or intercepted

Cc

wæ  
re  
re  
  
te:  
wi  
isi  
in;  
m'  
th

### 3.3.3 W



yo  
tic  
sig  
br  
  
all  
Hc  
es  
ar  
ar  
ve  
wæ  
næ  
di:  
  
ac  
ne  
in  
br  
re  
"W  
  
st:  
fo:  
co

wakeups. Although it might seem easier to use an interval time, you'd have to recompute it every time the thread wakes up, before waiting again—which would require determining how long it had already waited.

When a timed condition wait returns with the `ETIMEDOUT` error, you should test your predicate before treating the return as an error. If the condition for which you were waiting is true, the fact that it may have taken too long usually isn't important. Remember that a thread always relocks the mutex before returning from a condition wait, even when the wait times out. Waiting for a locked mutex after timeout can cause the timed wait to appear to have taken a lot longer than the time you requested.

### 3.3.3 Waking condition variable waiters

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Once you've got a thread waiting on a condition variable for some predicate, you'll probably want to wake it up. Pthreads provides two ways to wake a condition variable waiter. One is called "signal" and the other is called "broadcast." A signal operation wakes up a single thread waiting on the condition variable, while broadcast wakes up all threads waiting on the condition variable.

The term "signal" is easily confused with the "POSIX signal" mechanisms that allow you to define "signal actions," manipulate "signal masks," and so forth. However, the term "signal," as we use it here, had independently become well established in threading literature, and even in commercial implementations, and the Pthreads working group decided not to change the term. Luckily, there are few situations where we might be tempted to use both terms together—it is a very good idea to avoid using signals in threaded programs when at all possible. If we are careful to say "signal a condition variable" or "POSIX signal" (or "UNIX signal") where there is any ambiguity, we are unlikely to cause anyone severe discomfort.

It is easy to think of "broadcast" as a generalization of "signal," but it is more accurate to think of signal as an optimization of broadcast. Remember that it is never wrong to use broadcast instead of signal since waiters have to account for intercepted and spurious wakes. The only difference, in fact, is efficiency: A broadcast will wake additional threads that will have to test their predicate and resume waiting. But, in general, you can't replace a broadcast with a signal. "When in doubt, broadcast."

Use signal when only one thread needs to wake up to process the changed state, and when *any* waiting thread can do so. If you use one condition variable for several program predicate conditions, you can't use the signal operation; you couldn't tell whether it would awaken a thread waiting for that predicate, or for

another predicate. Don't try to get around that by resignaling the condition variable when you find the predicate isn't true. That might not pass on the signal as you expect; a spurious or intercepted wakeup could result in a series of pointless resignals.

If you add a single item to a queue, and only threads waiting for an item to appear are blocked on the condition variable, then you should probably use a signal. That'll wake up a single thread to check the queue and let the others sleep undisturbed, avoiding unnecessary context switches. On the other hand, if you add more than one item to the queue, you will probably need to broadcast. For examples of both broadcast and signal operations on condition variables, check out the "read/write lock" package in Section 7.1.2.

Although you must have the associated mutex locked to wait on a condition variable, you can signal (or broadcast) a condition variable with the associated mutex unlocked if that is more convenient. The advantage of doing so is that, on many systems, this may be more efficient. When a waiting thread awakens, it must first lock the mutex. If the thread awakens while the signaling thread holds the mutex, then the awakened thread must immediately block on the mutex—you've gone through two context switches to get back where you started.\*

Weighing on the other side is the fact that, if the mutex is not locked, any thread (not only the one being awakened) can lock the mutex prior to the thread being awakened. This race is one source of intercepted wakeups. A lower-priority thread, for example, might lock the mutex while another thread was about to awaken a very high-priority thread, delaying scheduling of the high-priority thread. If the mutex remains locked while signaling, this cannot happen—the high-priority waiter will be placed before the lower-priority waiter on the mutex, and will be scheduled first.

### 3.3.4 One final alarm program

It is time for one final version of our simple alarm program. In `alarm_mutex.c`, we reduced resource utilization by eliminating the use of a separate execution context (thread or process) for each alarm. Instead of separate execution contexts, we used a single thread that processed a list of alarms. There was one problem, however, with that approach—it was not responsive to new alarm commands. It had to finish waiting for one alarm before it could detect that another had been entered onto the list with an earlier expiration time, for example, if one entered the commands "10 message 1" followed by "5 message 2."

\*There is an optimization, which I've called "wait morphing," that moves a thread directly from the condition variable wait queue to the mutex wait queue in this case, without a context switch, when the mutex is locked. This optimization can produce a substantial performance benefit for many applications.

```

gr
al
ex
th
th
to
20,22
th
an
tin
va
iti

```

```

1 #i
2 #i
3 #i
4
5 /*
6 *
7 *
8 *
9 *
10 *
11 *
12 ty
13
14
15
16
17 }
18
19 pt
20 pt
21 al
22 ti

```

```

as
va
be
ak
en

```