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.

Now that we have added condition variables to our arsenal of threaded programming tools, we will solve that problem. The new version, creatively named `alarm_cond.c`, uses a timed condition wait rather than `sleep` to wait for an alarm expiration time. When `main` inserts a new entry at the head of the list, it signals the condition variable to awaken `alarm_thread` immediately. The `alarm_thread` then requeues the alarm on which it was waiting, to sort it properly with respect to the new entry, and tries again.

20,22 Part 1 shows the declarations for `alarm_cond.c`. There are two additions to this section, compared to `alarm_mutex.c`: a condition variable called `alarm_cond` and the `current_alarm` variable, which allows `main` to determine the expiration time of the alarm on which `alarm_thread` is currently waiting. The `current_alarm` variable is an optimization—`main` does not need to awaken `alarm_thread` unless it is either idle, or waiting for an alarm later than the one `main` has just inserted.

■ alarm_cond.c part 1 declarations

```
1  #include <pthread.h>
2  #include <time.h>
3  #include "errors.h"
4
5  /*
6   * The "alarm" structure now contains the time_t (time since the
7   * Epoch, in seconds) for each alarm, so that they can be
8   * sorted. Storing the requested number of seconds would not be
9   * enough, since the "alarm thread" cannot tell how long it has
10  * been on the list.
11  */
12 typedef struct alarm_tag {
13     struct alarm_tag    *link;
14     int                 seconds;
15     time_t              time;   /* seconds from EPOCH */
16     char                message[64];
17 } alarm_t;
18
19 pthread_mutex_t alarm_mutex = PTHREAD_MUTEX_INITIALIZER;
20 pthread_cond_t alarm_cond = PTHREAD_COND_INITIALIZER;
21 alarm_t *alarm_list = NULL;
22 time_t current_alarm = 0;
```

■ alarm_cond.c part 1 declarations

Part 2 shows the new function `alarm_insert`. This function is nearly the same as the list insertion code from `alarm_mutex.c`, except that it signals the condition variable `alarm_cond` when necessary. I made `alarm_insert` a separate function because now it needs to be called from two places—once by `main` to insert a new alarm, and now also by `alarm_thread` to reinsert an alarm that has been "preempted" by a new earlier alarm.

9-14 I have recommended that mutex locking protocols be documented, and here is an example: The `alarm_insert` function points out explicitly that it must be called with the `alarm_mutex` locked.

48-53 If `current_alarm` (the time of the next alarm expiration) is 0, then the `alarm_thread` is not aware of any outstanding alarm requests, and is waiting for new work. If `current_alarm` has a time greater than the expiration time of the new alarm, then `alarm_thread` is not planning to look for new work soon enough to handle the new alarm. In either case, signal the `alarm_cond` condition variable so that `alarm_thread` will wake up and process the new alarm.

■ alarm_cond.c part 2 alarm_insert

```
1  /*
2   * Insert alarm entry on list, in order.
3   */
4  void alarm_insert (alarm_t *alarm)
5  {
6      int status;
7      alarm_t **last, *next;
8
9      /*
10      * LOCKING PROTOCOL:
11      *
12      * This routine requires that the caller have locked the
13      * alarm_mutex!
14      */
15     last = &alarm_list;
16     next = *last;
17     while (next != NULL) {
18         if (next->time >= alarm->time) {
19             alarm->link = next;
20             *last = alarm;
21             break;
22         }
23         last = &next->link;
24         next = next->link;
25     }
26     /*
27      * If we reached the end of the list, insert the new alarm
28      * there.  ("next" is NULL, and "last" points to the link
29      * field of the last item, or to the list header.)
30      */
31     if (next == NULL) {
32         *last = alarm;
33         alarm->link = NULL;
34     }
35 #ifdef DEBUG
36     printf ("[list: ");
```

```
37      for (next = alarm_list; next != NULL; next = next->link)
38          printf ("%d(%d)[\"%s\"] ", next->time,
39              next->time - time (NULL), next->message);
40      printf ("]\n");
41  #endif
42      /*
43       * Wake the alarm thread if it is not busy (that is, if
44       * current_alarm is 0, signifying that it's waiting for
45       * work), or if the new alarm comes before the one on
46       * which the alarm thread is waiting.
47       */
48      if (current_alarm == 0 || alarm->time < current_alarm) {
49          current_alarm = alarm->time;
50          status = pthread_cond_signal (&alarm_cond);
51          if (status != 0)
52              err_abort (status, "Signal cond");
53      }
54  }
```

■ alarm_cond.c part 2 alarm_insert

Part 3 shows the `alarm_thread` function, the start function for the "alarm server" thread. The general structure of `alarm_thread` is very much like the `alarm_thread` in `alarm_mutex.c`. The differences are due to the addition of the condition variable.

26-31 If the `alarm_list` is empty, `alarm_mutex.c` could do nothing but `sleep` anyway, so that `main` would be able to process a new command. The result was that it could not see a new alarm request for at least a full second. Now, `alarm_thread` instead waits on the `alarm_cond` condition variable, with no timeout. It will "sleep" until you enter a new alarm command, and then `main` will be able to awaken it immediately. Setting `current_alarm` to 0 tells `main` that `alarm_thread` is idle. Remember that `pthread_cond_wait` unlocks the mutex before waiting, and relocks the mutex before returning to the caller.

35 The new variable `expired` is initialized to 0; it will be set to 1 later if the timed condition wait expires. This makes it a little easier to decide whether to print the current alarm's message at the bottom of the loop.

36-42 If the alarm we've just removed from the list hasn't already expired, then we need to wait for it. Because we're using a timed condition wait, which requires a POSIX.1b `struct timespec`, rather than the simple integer time required by `sleep`, we convert the expiration time. This is easy, because a `struct timespec` has two members—`tv_sec` is the number of seconds since the Epoch, which is exactly what we already have from the `time` function, and `tv_nsec` is an additional count of nanoseconds. We will just set `tv_nsec` to 0, since we have no need of the greater resolution.

43 Record the expiration time in the `current_alarm` variable so that `main` can determine whether to signal `alarm_cond` when a new alarm is added.

44-53 Wait until either the current alarm has expired, or `main` requests that `alarm_thread` look for a new, earlier alarm. Notice that the predicate test is split here, for convenience. The expression in the while statement is only half the predicate, detecting that `main` has changed `current_alarm` by inserting an earlier timer. When the timed wait returns `ETIMEDOUT`, indicating that the current alarm has expired, we exit the while loop with a `break` statement at line 49.

54-55 If the while loop exited when the current alarm had not expired, `main` must have asked `alarm_thread` to process an earlier alarm. Make sure the current alarm isn't lost by reinserting it onto the list.

57 If we remove from `alarm_list` an alarm that has already expired, just set the `expired` variable to 1 to ensure that the message is printed.

■ alarm_cond.c part 3 alarm_routine

```
 1 /*
 2  * The alarm thread's start routine.
 3  */
 4 void *alarm_thread (void *arg)
 5 {
 6     alarm_t *alarm;
 7     struct timespec cond_time;
 8     time_t now;
 9     int status, expired;
10
11     /*
12      * Loop forever, processing commands. The alarm thread will
13      * be disintegrated when the process exits. Lock the mutex
14      * at the start -- it will be unlocked during condition
15      * waits, so the main thread can insert alarms.
16      */
17     status = pthread_mutex_lock (&alarm_mutex);
18     if (status != 0)
19         err_abort (status, "Lock mutex");
20     while (1) {
21         /*
22          * If the alarm list is empty, wait until an alarm is
23          * added. Setting current_alarm to 0 informs the insert
24          * routine that the thread is not busy.
25          */
26         current_alarm = 0;
27         while (alarm_list == NULL) {
28             status = pthread_cond_wait (&alarm_cond, &alarm_mutex);
29             if (status != 0)
30                 err_abort (status, "Wait on cond");
31             }
32         alarm = alarm_list;
33         alarm_list = alarm->link;
34         now = time (NULL);
```

```
35          expired = 0;
36          if (alarm->time > now) {
37  #ifdef DEBUG
38              printf ("[waiting: %d(%d)\"%s\"]\n", alarm->time,
39                  alarm->time - time (NULL), alarm->message);
40  #endif
41              cond_time.tv_sec = alarm->time;
42              cond_time.tv_nsec = 0;
43              current_alarm = alarm->time;
44              while (current_alarm == alarm->time) {
45                  status = pthread_cond_timedwait (
46                      &alarm_cond, &alarm_mutex, &cond_time);
47                  if (status == ETIMEDOUT) {
48                      expired = 1;
49                      break;
50                  }
51                  if (status != 0)
52                      err_abort (status, "Cond timedwait");
53              }
54              if (!expired)
55                  alarm_insert (alarm);
56          } else
57              expired = 1;
58          if (expired) {
59              printf ("(%d) %s\n", alarm->seconds, alarm->message);
60              free (alarm);
61          }
62      }
63  }
```

■ alarm_cond.c part 3 alarm_routine

Part 4 shows the final section of `alarm_cond.c`, the `main` program. It is nearly identical to the `main` function from `alarm_mutex.c`.

38 Because the condition variable signal operation is built into the new `alarm_insert` function, we call `alarm_insert` rather than inserting a new alarm directly.

■ alarm_cond.c part 4 main

```
1  int main (int argc, char *argv[])
2  {
3      int status;
4      char line[128];
5      alarm_t *alarm;
6      pthread_t thread;
7
8      status = pthread_create (
9          &thread, NULL, alarm_thread, NULL);
```

```
10      if (status != 0)
11          err_abort (status, "Create alarm thread");
12      while (1) {
13          printf ("Alarm> ");
14          if (fgets (line, sizeof (line), stdin) == NULL) exit (0);
15          if (strlen (line) <= 1) continue;
16          alarm = (alarm_t*)malloc (sizeof (alarm_t));
17          if (alarm == NULL)
18              errno_abort ("Allocate alarm");
19
20          /*
21           * Parse input line into seconds (%d) and a message
22           * (%64[^\n]), consisting of up to 64 characters
23           * separated from the seconds by whitespace.
24           */
25          if (sscanf (line, "%d %64[^\n]",
26              &alarm->seconds, alarm->message) < 2) {
27              fprintf (stderr, "Bad command\n");
28              free (alarm);
29          } else {
30              status = pthread_mutex_lock (&alarm_mutex);
31              if (status != 0)
32                  err_abort (status, "Lock mutex");
33              alarm->time = time (NULL) + alarm->seconds;
34              /*
35               * Insert the new alarm into the list of alarms,
36               * sorted by expiration time.
37               */
38              alarm_insert (alarm);
39              status = pthread_mutex_unlock (&alarm_mutex);
40              if (status != 0)
41                  err_abort (status, "Unlock mutex");
42          }
43      }
44  }
```

■ alarm_cond.c part 4 main

3.4 Memory visibility between threads

The moment Alice appeared, she was appealed to by all three to settle the question, and they repeated their arguments to her, though, as they all spoke at once, she found it very hard to make out exactly what they said.

—Lewis Carroll, Alice's Adventures in Wonderland