3.2.2 Locking and unlocking a mutex

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

In the simplest case, using a mutex is easy. You lock the mutex by calling either `pthread_mutex_lock` or `pthread_mutex_trylock`, do something with the shared data, and then unlock the mutex by calling `pthread_mutex_unlock`. To make sure that a thread can read consistent values for a series of variables, you need to lock your mutex around any section of code that reads *or* writes those variables.

You cannot lock a mutex when the calling thread already has that mutex locked. The result of attempting to do so may be an error return, or it may be a self-deadlock, with the unfortunate thread waiting forever for itself to unlock the mutex. (If you have access to a system supporting the UNIX98 thread extensions, you can create mutexes of various types, including *recursive* mutexes, which allow a thread to relock a mutex it already owns. The mutex *type* attribute is discussed in Section 10.1.2.)

The following program, `alarm_mutex.c`, is an improved version of `alarm_thread.c` (from Chapter 1). It lines up multiple alarm requests in a single "alarm server" thread.

12-17 The `alarm_t` structure now contains an absolute time, as a standard UNIX `time_t`, which is the number of seconds from the UNIX Epoch (Jan 1 1970 00:00) to the expiration time. This is necessary so that `alarm_t` structures can be sorted by "expiration time" instead of merely by the requested number of seconds. In addition, there is a `link` member to connect the list of alarms.

19-20 The `alarm_mutex` mutex coordinates access to the list head for alarm requests, called `alarm_list`. The mutex is statically initialized using default attributes, with the `PTHREAD_MUTEX_INITIALIZER` macro. The list head is initialized to `NULL`, or empty.

■ alarm_mutex.c part 1 definitions

```
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  alarm_t *alarm_list = NULL;
```

■ alarm_mutex.c part 1 definitions

The code for the `alarm_thread` function follows. This function is run as a thread, and processes each alarm request in order from the list `alarm_list`. The thread never terminates—when `main` returns, the thread simply "evaporates." The only consequence of this is that any remaining alarms will not be delivered—the thread maintains no state that can be seen outside the process.

If you would prefer that the program process all outstanding alarm requests before exiting, you can easily modify the program to accomplish this. The main thread must notify `alarm_thread`, by some means, that it should terminate when it finds the `alarm_list` empty. You could, for example, have `main` set a new global variable `alarm_done` and then terminate using `pthread_exit` rather than `exit`. When `alarm_thread` finds `alarm_list` empty and `alarm_done` set, it would immediately call `pthread_exit` rather than waiting for a new entry.

29-30 If there are no alarms on the list, `alarm_thread` needs to block itself, with the mutex unlocked, at least for a short time, so that `main` will be able to add a new alarm. It does this by setting `sleep_time` to one second.

31-42 If an alarm is found, it is removed from the list. The current time is retrieved by calling the `time` function, and it is compared to the requested time for the alarm. If the alarm has already expired, then `alarm_thread` will set `sleep_time` to 0. If the alarm has not expired, `alarm_thread` computes the difference between the current time and the alarm expiration time, and sets `sleep_time` to that number of seconds.

52-58 The mutex is always unlocked before sleeping or yielding. If the mutex remained locked, then `main` would be unable to insert a new alarm on the list. That would make the program behave synchronously—the user would have to wait until the alarm expired before doing anything else. (The user would be able to enter a single command, but would not receive another prompt until the next alarm expired.) Calling sleep blocks `alarm_thread` for the required period of time—it cannot run until the timer expires.

Calling `sched_yield` instead is slightly different. We'll describe `sched_yield` in detail later (in Section 5.5.2)—for now, just remember that calling `sched_yield` will yield the processor to a thread that is ready to run, but will return immediately if there are no *ready* threads. In this case, it means that the main thread will be allowed to process a user command if there's input waiting—but if the user hasn't entered a command, `sched_yield` will return immediately.

64-67 If the `alarm` pointer is not `NULL`, that is, if an alarm was processed from `alarm_list`, the function prints a message indicating that the alarm has expired. After printing the message, it frees the alarm structure. The thread is now ready to process another alarm.

alarm_mutex.c part 2 alarm_thread

```
1  /*
2   * The alarm thread's start routine.
3   */
4  void *alarm_thread (void *arg)
5  {
6      alarm_t *alarm;
7      int sleep_time;
8      time_t now;
9      int status;
10
11     /*
12      * Loop forever, processing commands. The alarm thread will
13      * be disintegrated when the process exits.
14      */
15     while (1) {
16         status = pthread_mutex_lock (&alarm_mutex);
17         if (status != 0)
18             err_abort (status, "Lock mutex");
19         alarm = alarm_list;
20
21         /*
22          * If the alarm list is empty, wait for one second. This
23          * allows the main thread to run, and read another
24          * command. If the list is not empty, remove the first
25          * item. Compute the number of seconds to wait -- if the
26          * result is less than 0 (the time has passed), then set
27          * the sleep_time to 0.
28          */
29         if (alarm == NULL)
30             sleep_time = 1;
31         else {
32             alarm_list = alarm->link;
33             now = time (NULL);
34             if (alarm->time <= now)
35                 sleep_time = 0;
36             else
37                 sleep_time = alarm->time - now;
38 #ifdef DEBUG
39             printf ("[waiting: %d(%d)\"%s\"]\n", alarm->time,
40                 sleep_time, alarm->message);
41 #endif
42             }
43
```

```
44          /*
45           * Unlock the mutex before waiting, so that the main
46           * thread can lock it to insert a new alarm request. If
47           * the sleep_time is 0, then call sched_yield, giving
48           * the main thread a chance to run if it has been
49           * readied by user input, without delaying the message
50           * if there's no input.
51           */
52          status = pthread_mutex_unlock (&alarm_mutex);
53          if (status != 0)
54              err_abort (status, "Unlock mutex");
55          if (sleep_time > 0)
56              sleep (sleep_time);
57          else
58              sched_yield ();
59
60          /*
61           * If a timer expired, print the message and free the
62           * structure.
63           */
64          if (alarm != NULL) {
65              printf ("(%d) %s\n", alarm->seconds, alarm->message);
66              free (alarm);
67          }
68      }
69  }
```

■ alarm_mutex.c part 2 alarm_thread

And finally, the code for the main program for `alarm_mutex.c`. The basic structure is the same as all of the other versions of the alarm program that we've developed—a loop, reading simple commands from `stdin` and processing each in turn. This time, instead of waiting synchronously as in `alarm.c`, or creating a new asynchronous entity to process each alarm command as in `alarm_fork.c` and `alarm_thread.c`, each request is queued to a server thread, `alarm_thread`. As soon as `main` has queued the request, it is free to read the next command.

8-11 Create the server thread that will process all alarm requests. Although we don't use it, the thread's ID is returned in local variable `thread`.

13-28 Read and process a command, much as in any of the other versions of our alarm program. As in `alarm_thread.c`, the data is stored in a heap structure allocated by `malloc`.

30-32 The program needs to add the alarm request to `alarm_list`, which is shared by both `alarm_thread` and `main`. So we start by locking the mutex that synchronizes access to the shared data, `alarm_mutex`.

33 Because `alarm_thread` processes queued requests, serially, it has no way of knowing how much time has elapsed between reading the command and processing it. Therefore, the alarm structure includes the absolute time of the alarm expiration, which we calculate by adding the alarm interval, in seconds, to the

current number of seconds since the UNIX Epoch, as returned by the `time` function.

39-49 The alarms are sorted in order of expiration time on the `alarm_list` queue. The insertion code searches the queue until it finds the first entry with a time greater than or equal to the new alarm's time. The new entry is inserted preceding the located entry. Because `alarm_list` is a simple linked list, the traversal maintains a current entry pointer (`this`) and a pointer to the previous entry's `link` member, or to the `alarm_list` head pointer (`last`).

56-59 If no alarm with a time greater than or equal to the new alarm's time is found, then the new alarm is inserted at the end of the list. That is, if the alarm pointer is `NULL` on exit from the search loop (the last entry on the list always has a `link` pointer of `NULL`), the previous entry (or queue head) is made to point to the new entry.

■ alarm_mutex.c part 3 main

```
1  int main (int argc, char *argv[])
2  {
3      int status;
4      char line[128];
5      alarm_t *alarm, **last, *next;
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              /*
36               * Insert the new alarm into the list of alarms,
37               * sorted by expiration time.
38               */
39              last = &alarm_list;
40              next = *last;
41              while (next != NULL) {
42                  if (next->time >= alarm->time) {
43                      alarm->link = next;
44                      *last = alarm;
45                      break;
46                  }
47                  last = &next->link;
48                  next = next->link;
49              }
50              /*
51               * If we reached the end of the list, insert the new
52               * alarm there. ("next" is NULL, and "last" points
53               * to the link field of the last item, or to the
54               * list header).
55               */
56              if (next == NULL) {
57                  *last = alarm;
58                  alarm->link = NULL;
59              }
60  #ifdef DEBUG
61              printf ("[list: ");
62              for (next = alarm_list; next != NULL; next = next->link)
63                  printf ("%d(%d)[\"%s\"] ", next->time,
64                      next->time - time (NULL), next->message);
65              printf ("]\n");
66  #endif
67              status = pthread_mutex_unlock (&alarm_mutex);
68              if (status != 0)
69                  err_abort (status, "Unlock mutex");
70          }
71      }
72  }
```

■ alarm_mutex.c part 3 main

This simple program has a few severe failings. Although it has the advantage, compared to `alarm_fork.c` or `alarm_thread.c`, of using fewer resources, it is less responsive. Once `alarm_thread` has accepted an alarm request from the queue, it

sleeps until that alarm expires. When it fails to find an alarm request on the list, it sleeps for a second anyway, to allow `main` to accept another alarm command. During all this sleeping, it will fail to notice any alarm requests added to the head of the queue by `main`, until it returns from `sleep`.

This problem could be addressed in various ways. The simplest, of course, would be to go back to `alarm_thread.c`, where a thread was created for each alarm request. That wasn't so bad, since threads are relatively cheap. They're still not as cheap as the `alarm_t` data structure, however, and we'd like to make efficient programs—not just responsive programs. The best solution is to make use of condition variables for signaling changes in the state of shared data, so it shouldn't be a surprise that you'll be seeing one final version of the alarm program, `alarm_cond.c`, in Section 3.3.4.

3.2.2.1 Nonblocking mutex locks

When you lock a mutex by calling `pthread_mutex_lock`, the calling thread will block if the mutex is already locked. Normally, that's what you want. But occasionally you want your code to take some alternate path if the mutex is locked. Your program may be able to do useful work instead of waiting. Pthreads provides the `pthread_mutex_trylock` function, which will return an error status (`EBUSY`) instead of blocking if the mutex is already locked.

When you use a nonblocking mutex lock, be careful to *unlock* the mutex only if `pthread_mutex_trylock` returned with success status. Only the thread that owns a mutex may unlock it. An erroneous call to `pthread_mutex_unlock` may return an error, or it may unlock the mutex while some other thread relies on having it locked—and that will probably cause your program to break in ways that may be very difficult to debug.

The following program, `trylock.c`, uses `pthread_mutex_trylock` to occasionally report the value of a counter—but only when its access does not conflict with the counting thread.

4 This definition controls how long `counter_thread` holds the mutex while updating the `counter`. Making this number larger increases the chance that the `pthread_mutex_trylock` in `monitor_thread` will occasionally return `EBUSY`.

14-39 The `counter_thread` wakes up approximately each second, locks the mutex, and spins for a while, incrementing `counter`. The `counter` is therefore increased by `SPIN` each second.

46-72 The `monitor_thread` wakes up every three seconds, and tries to lock the mutex. If the attempt fails with `EBUSY`, `monitor_thread` counts the failure and waits another three seconds. If the `pthread_mutex_trylock` succeeds, then `monitor_thread` prints the current value of `counter` (scaled by `SPIN`).

80-88 On Solaris 2.5, call `thr_setconcurrency` to set the thread concurrency level to 2. This allows the `counter_thread` and `monitor_thread` to run concurrently on a uniprocessor. Otherwise, `monitor_thread` would not run until `counter_thread` terminated.