Fall 2004 Handout 9

Pthreads

The purpose of threads is to avoid **`fork/exec/wait`** and semaphores:

Unix system calls	*Pthreads*
`fork`	`pthread_create`
`exec`	
`waitpid`	`pthread_join`
`exit`	`pthread_exit`
`kill(,SIGKILL)`	`pthread_cancel`
`getpid`	`pthread_self`
`semget(IPC_CREAT`	`pthread_mutex_init`
`semop(decrement`	`pthread_mutex_lock`
`semop(increment`	`pthread_mutex_unlock`
`semctl(IPC_RMID`	`pthread_mutex_destroy`

See *Pthreads Programming* by Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell; O'Reilly & Associates, 1996; ISBN 1-56592-115-1.

`http://www.oreilly.com/catalog/pthread/`

Pthreads are like processes, except

(1) They're faster to create, destroy, and have take turns because the operating system doesn't necessarily know about them (but see the **`-L`** option of **`ps`**; **`-m`** on other systems).

(2) There is no concept of ''parent'' and ''child'': any thread in a process can wait for (or cause) the death of any other thread in the process. The original thread is informally called the *main thread* (pp. 13, 15 in the Pthreads book).

(3) A child executes the body of an **`if`** statement. A thread executes the body of a function, called the thread's *start routine*. The start routine can take only one argument, which must be a pointer to **`void`**. If you need more arguments, put them into a structure and pass a pointer to the structure.

(4) All the threads of a process share the same global variables.

When does a thread die?

(1) A process dies by calling **`exit`**, or by **`return`**'ing from **`main`**, or by receiving a fatal signal (e.g., **`SIGKILL`**). When the process dies, all the threads inside of it die with it.

(2) A thread dies by calling **`pthread_exit`**, or by **`return`**'ing from its start routine. (Another thread can wait for it to die by calling **`pthread_join`**. If no one will call **`pthread_join`**, call **`pthread_detach`** so that the zombie thread will disappear immediately.)

(3) One thread can kill another by calling **`pthread_cancel`**. The main thread cannot be cancelled because it has no **`pthread_t`** to be the first argument of **`pthread_cancel`**.

An echo client, with two threads instead of two processes

—Source code on the Web at **`http://i5.nyu.edu/~mm64/x52.9544/src/pthread1.c`**

```
1 #include <stdio.h>
```



```
 2 #include <stdlib.h>
 3 #include <unistd.h>   /* for getpid */
 4 #include <string.h>
 5 #include <pthread.h>
 6 #include <errno.h>     /* for ESRCH */
 7
 8 #include <sys/types.h>
 9 #include <sys/socket.h>
10 #include <netinet/in.h>
11
12 char *progname;
13 void *thread_function(void *s);
14
15 int main(int argc, char **argv)
16 {
17     const int s = socket(AF_INET, SOCK_STREAM, 0);
18     struct sockaddr_in address;
19     char buffer[INET_ADDRSTRLEN]; /* for inet_ntop */
20     pthread_t thread;
21     int retval;                   /* instead of errno */
22     char c;                       /* write requires a char: KR p. 170 */
23
24     progname = argv[0];
25     if (s < 0) {
26         perror(progname);
27         return 1;
28     }
29
30     bzero((char *)&address, sizeof address);
31     address.sin_family = AF_INET;
32     address.sin_port = htons(7);
33
34     /* IP address for smail.let.uu.nl: */
35     retval = inet_pton(AF_INET, "131.211.194.40", &address.sin_addr);
36     if (retval == 0) {
37         fprintf(stderr, "%s: bad dotted string to inet_pton\n", argv[0]);
38         return 2;
39     } else if (retval != 1) {
40         perror(argv[0]);
41         return 3;
42     }
43
44     if (inet_ntop(AF_INET, &address.sin_addr, buffer, sizeof buffer) == NULL) {
45         perror(argv[0]);
46         return 4;
47     }
48     printf("Trying %s...\n", buffer);
49
50     if (connect(s, (struct sockaddr *)&address, sizeof address) != 0) {
51         perror(progname);
52         return 5;
53     }
54
55     printf("Connected to smail.let.uu.nl.\n"
```



```
 56          "Escape character is '^d'.\n");
 57
 58      if (pthread_create(&thread, NULL, thread_function, (void *)&s) != 0) {
 59          perror(progname);
 60          return 6;
 61      }
 62
 63      retval = pthread_detach(thread);
 64      if (retval != 0) {
 65          fprintf(stderr, "%s: pthread_detach returned %d.\n", argv[0], retval);
 66          return 7;
 67      }
 68
 69      sprintf(buffer, "ps -Lf -p %d", getpid());
 70      system(buffer);
 71
 72      /*
 73      The main thread will copy from the server to the stdout.
 74      */
 75      while (read(s, &c, 1) == 1) {
 76          putchar(c);
 77      }
 78
 79      /*
 80      Arrive here after receiving EOF from the server.
 81      */
 82      fprintf(stderr, "Connection closed by foreign host.\n");
 83
 84      /*
 85      If pthread_cancel returns ESRCH, it means that the thread has already
 86      died before we had a chance to cancel it.
 87      */
 88      retval = pthread_cancel(thread);
 89      if (retval != 0 && retval != ESRCH) {
 90          fprintf(stderr, "%s: pthread_cancel returned %d.\n", argv[0], retval);
 91          return 8;
 92      }
 93
 94      if (shutdown(s, SHUT_RDWR) != 0) {
 95          perror(progname);
 96          return 9;
 97      }
 98
 99      return EXIT_SUCCESS;
100  }
101
102  void *thread_function(void *s)
103  {
104      const int fd = *(int *)s;
105      int i;
106
107      /*
108      This thread will copy from the stdin to the server.
109      */
```

```
110     while ((i = getchar()) != EOF) {
111         const char c = i;
112         write(fd, &c, 1);
113     }
114
115     if (shutdown(fd, SHUT_WR) != 0) {/* close socket for writing */
116         perror(progname);
117         exit(10);
118     }
119 }
```

```
3$ gcc -o ~/bin/pthread1 pthread1.c -lsocket -lnsl -lpthread
4$ ls -l ~/bin/pthread1

5$ pthread1
Trying 131.211.194.40...
Connected to smail.let.uu.nl.
Escape character is '^d'.
     UID   PID  PPID   LWP  NLWP  C    STIME TTY      LTIME CMD
    mm64 24875 23445     1     4  0 15:48:24 pts/2    0:00 pthread1
    mm64 24875 23445     2     4  0 15:48:24 pts/2    0:00 pthread1
    mm64 24875 23445     3     4  0 15:48:24 pts/2    0:00 pthread1
    mm64 24875 23445     4     4  0 15:48:24 pts/2    0:00 pthread1
```

hello

hello

goodbye

goodbye

control-d

```
Connection closed by foreign host.

6$ echo $?
0
```

Also try connecting to port 13 (**daytime**) of 129.206.218.89 (**www.urz.uni-heidelberg.de**).

Serve more than one client simultaneously with threads

The above program created exactly one thread, so all we had to do was declare exactly one **pthread_t** (line 19) and exactly one **s** (line 16) The following program creates an unpredictable number of threads, so we have to use **malloc** and **free** to manufacture each thread's argument.

—Source code on the Web at **http://i5.nyu.edu/~mm64/x52.9544/src/pthread2.c**

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <pthread.h>
 5
 6 #include <sys/types.h>
 7 #include <sys/socket.h>
 8 #include <netinet/in.h>
 9
10 char *progname;
11 void *thread_function(void *s);
12
13 int main(int argc, char **argv)
```

```
14 {
15     const int s = socket(AF_INET, SOCK_STREAM, 0);
16     struct sockaddr_in myaddress;
17     struct sockaddr_in clientaddress;
18     socklen_t length = sizeof clientaddress;
19 
20     progname = argv[0];
21     if (s < 0) {
22         perror(progname);
23         return 1;
24     }
25 
26     bzero((char *)&myaddress, sizeof myaddress);
27     myaddress.sin_family = AF_INET;
28     myaddress.sin_port = htons(9266);
29     myaddress.sin_addr.s_addr = INADDR_ANY;
30 
31     if (bind(s, (const struct sockaddr *)&myaddress, sizeof myaddress) != 0) {
32         perror(progname);
33         return 2;
34     }
35 
36     if (listen(s, SOMAXCONN) != 0) {
37         perror(progname);
38         return 3;
39     }
40 
41     for (;;) {
42         pthread_t thread;
43         int retval;
44 
45         /*
46         file descriptor for talking to each new client
47         */
48         int *const pclient = malloc(sizeof (int));
49         if (pclient == NULL) {
50             perror(progname);
51             return 4;
52         }
53 
54         *pclient = accept(s, (struct sockaddr *)&clientaddress, &length);
55         if (*pclient < 0) {
56             perror(progname);
57             return 5;
58         }
59 
60         retval = pthread_create(&thread, NULL, thread_function, pclient);
61         if (retval != 0) {
62             fprintf(stderr, "%s: pthread_create returned %d.\n",
63                 argv[0], retval);
64             return 6;
65         }
66 
67         retval = pthread_detach(thread);
```

```
 68         if (retval) {
 69             fprintf(stderr, "%s: pthread_detach returned %d.\n",
 70                 argv[0], retval);
 71             return 7;
 72         }
 73     }
 74 }
 75
 76 void *thread_function(void *s)
 77 {
 78     const int client = *(int *)s;
 79     char buffer[2];
 80
 81     /*
 82     The service provided by this server is merely to input one character
 83     and then output it in lowercase, followed by a newline.
 84     */
 85
 86     if (read(client, buffer, 1) != 1) {
 87         perror(progname);
 88         exit(8);
 89     }
 90
 91     buffer[0] = tolower(buffer[0]);
 92     buffer[1] = '\n';
 93
 94     if (write(client, buffer, sizeof buffer) != sizeof buffer) {
 95         perror(progname);
 96         exit(9);
 97     }
 98
 99     if (shutdown(client, SHUT_RDWR) != 0) {
100         perror(progname);
101         exit(10);
102     }
103
104     free(s);
105 }
```

```
1$ gcc -o ~/bin/pthread2 pthread2.c -lsocket -lnsl -lpthread
2$ ls -l ~/bin/pthread2
3$ pthread2 &
4$ netstat -an | grep 9266

5$ telnet i5.nyu.edu 9266
Trying 128.122.108.193...
Connected to i5.nyu.edu.
Escape character is '^]'.
A
a
Connection closed by foreign host.
```


Protect critical sections with a mutex variable

Here's a program that creates one thread. The main thread and the created thread will then both increment the global **`int n`**.

—Source code on the Web at **`http://i5.nyu.edu/~mm64/x52.9544/src/mutex.c`**

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <pthread.h>
 4
 5 void *thread_function(void *p);
 6
 7 const char *progname;
 8 int n = 0;
 9 pthread_mutex_t mutex;   /* protects n */
10
11 int main(int argc, char **argv)
12 {
13     pthread_t thread;
14     void *status;
15     int retval;
16
17     progname = argv[0];
18
19     retval = pthread_mutex_init(&mutex, NULL);
20     if (retval != 0) {
21         fprintf(stderr, "%s: pthread_mutex_init returned %d.\n",
22             argv[0], retval);
23         return 1;
24     }
25
26     if (pthread_create(&thread, NULL,
27         thread_function, &n) != 0) {
28         perror(progname);
29         return 2;
30     }
31
32     if (pthread_mutex_lock(&mutex) != 0) {
33         perror(progname);
34         return 3;
35     }
36     /* Start of critical section. */
37
38     ++n;
39
40     /* End of critical section. */
41     if (pthread_mutex_unlock(&mutex) != 0) {
42         perror(progname);
43         return 4;
44     }
45
46     if (pthread_join(thread, &status) != 0) {     /* like waitpid */
47         perror(progname);
48         return 5;
49     }
```

```
50
51      if (pthread_mutex_destroy(&mutex) < 0) {
52          perror(progname);
53          return 6;
54      }
55
56      printf("n == %d, and the thread returned \"%s\".\n",
57          n, (char *)status);
58  }
59
60  void *thread_function(void *p)
61  {
62      if (pthread_mutex_lock(&mutex) != 0) {
63          perror(progname);
64          exit(7);
65      }
66      /* Start of critical section. */
67
68      ++*(int *)p;
69
70      /* End of critical section. */
71      if (pthread_mutex_unlock(&mutex) != 0) {
72          perror(progname);
73          exit(8);
74      }
75
76      pthread_exit("goodbye");
77  }
```

```
n == 2 and the thread returned "goodbye".
```

Wait until something happens

The main thread should wait until the variable **count** assumes the value **n**.

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <pthread.h>
 4
 5  const int n = 10;
 6  int count = 0;
 7  pthread_mutex_t mutex;  /* protects count */
 8  char *progname;
 9
10  void thread_function(void *p);
11
12  int main(int argc, char **argv)
13  {
14      pthread_t thread;
15      int keep_looping = 1;
16
17      progname = argv[0];
18
19      if (pthread_mutex_init(&mutex, pthread_mutexattr_default) != 0) {
20          perror(progname);
```

```
21          return 1;
22      }
23
24      if (pthread_create(&thread, pthread_attr_default,
25          (pthread_startroutine_t)thread_function, NULL) != 0) {
26          perror(progname);
27          return 2;
28      }
29
30      if (pthread_detach(&thread) != 0) {
31          perror(progname);
32          return 3;
33      }
34
35      /* Wait until count == n. */
36      while (keep_looping) {
37          if (pthread_mutex_lock(&mutex) != 0) {
38              perror(progname);
39              return 4;
40          }
41          /* Start of critical section. */
42
43          if (count == n) {
44              keep_looping = 0;
45          }
46
47          /* End of critical section. */
48          if (pthread_mutex_unlock(&mutex) != 0) {
49              perror(progname);
50              return 5;
51          }
52      }
53
54      if (pthread_mutex_destroy(&mutex) != 0) {
55          perror(progname);
56          return 6;
57      }
58
59      printf ("count has reached %d.\n", n);
60      return EXIT_SUCCESS;
61  }
62
63  void thread_function(void *p)
64  {
65      int i;
66
67      for (i = 0; i < n; ++i) {
68          if (pthread_mutex_lock(&mutex) != 0) {
69              perror(progname);
70              exit(7);
71          }
72          /* Start of critical section. */
73
74          ++count;
```

```
75
76          /* End of critical section. */
77          if (pthread_mutex_unlock(&mutex) != 0) {
78              perror(progname);
79              exit(8);
80          }
81      }
82
83      for (;;) {
84          /* lots of other work */
85      }
86  }
```

```
count has reached 10.
```

Wait until something happens, using condition variables

The main thread should wait until the variable **count** assumes the value **n**.

Write a logical expression (called a *predicate*) in a comment alongside the declaration of a condition variable (line 8). The condition variable will wake all the threads that are waiting for the predicate to become true.

The predicate will usually contain another variable (in this case, **count**) that will be used by more than one thread. **count** must therefore be protected by a **mutex** to make sure that only at most one thread at a time will access it. Whenever you have a condition variable, you therefore usually also need a **mutex**: the condition variable alone is not enough.

The **pthread_cond_wait** in line 48 takes pointers to a condition variable and the **mutex** that protects the variable in the condition variable's predicate. The **mutex** must already be locked (line 41). **pthread_cond_wait** unlocks the **mutex** and then waits until some other thread calls **pthread_cond_signal**. (If **pthread_cond_wait** didn't unlock the **mutex**, then no other thread could change **count** and **pthread_cond_wait** would wait forever.)

When the predicate finally becomes true (line 85), some other thread calls **pthread_cond_signal** in line 86. Note that the other thread locks the **mutex** (line 79) before calling **pthread_cond_signal**, and unlocks the mutex afterward (line 93).

After the **pthread_cond_signal** is called in line 86 and the **mutex** is unlocked in line 93, the **pthread_cond_wait** locks the **mutex** again and returns. To sum up: the **mutex** must be locked before the call to **pthread_cond_wait** in line 86, and is guaranteed to be locked when we return from **pthread_cond_wait**, but is unlocked and locked again during the call.

One final complication. After the **pthread_cond_signal** is called in line 86 and the **mutex** is unlocked in line 93, but before the **pthread_cond_wait** locks the **mutex** again and returns, it is possible for some other thread to quickly lock the **mutex**, decrement **count**, and **unlock** the **mutex**. When we return from **pthread_cond_wait**, we may therefore find that the predicate we've been waiting for, **count == n**, is false. In this case, we must call **pthread_cond_wait** again, and as many times as necessary. Therefore change the **if** to **while** in line 47. (This is also necessary because of *spurious wakeups*.)

41 lock
48 unlock
    79 lock
    86 signal
    93 unlock
48 lock
55 unlock


```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <pthread.h>
 4
 5 const int n = 10;
 6 int count = 0;
 7 pthread_mutex_t mutex;        /* protects count */
 8 pthread_cond_t condition;     /* count == n */
 9 char *progname;
10
11 void *thread_function(void *p);
12
13 int main(int argc, char **argv)
14 {
15     pthread_t thread;
16
17     progname = argv[0];
18
19     if (pthread_mutex_init(&mutex, pthread_mutexattr_default) != 0) {
20         perror(progname);
21         return 1;
22     }
23
24     if (pthread_cond_init(&condition, pthread_condattr_default) != 0) {
25         perror(progname);
26         return 2;
27     }
28
29     if (pthread_create(&thread, pthread_attr_default,
30         (pthread_startroutine_t)thread_function, NULL) != 0) {
31         perror(progname);
32         return 3;
33     }
34
35     if (pthread_detach(&thread) != 0) {
36         perror(progname);
37         return 4;
38     }
39
40     /* Wait until count == n. */
41     if (pthread_mutex_lock(&mutex) != 0) {
42         perror(progname);
43         return 5;
44     }
45     /* Start of critical section. */
46
47     if (count != n) {               /* change "if" to "while" */
48         if (pthread_cond_wait(&condition, &mutex) != 0) {
49             perror(progname);
50             return 6;
51         }
52     }
53
54     /* End of critical section. */
```

```
 55     if (pthread_mutex_unlock(&mutex) != 0) {
 56         perror(progname);
 57         return 7;
 58     }
 59
 60     if (pthread_cond_destroy(&condition) != 0) {
 61         perror(progname);
 62         return 8;
 63     }
 64
 65     if (pthread_mutex_destroy(&mutex) != 0) {
 66         perror(progname);
 67         return 9;
 68     }
 69
 70     printf ("count has reached %d.\n", n);
 71     return EXIT_SUCCESS;
 72 }
 73
 74 void *thread_function(void *p)
 75 {
 76     int i;
 77
 78     for (i = 0; i < n; ++i) {
 79         if (pthread_mutex_lock(&mutex) != 0) {
 80             perror(progname);
 81             exit(10);
 82         }
 83         /* Start of critical section. */
 84
 85         if (++count == n) {
 86             if (pthread_cond_signal(&condition) != 0) {
 87                 perror(progname);
 88                 exit(11);
 89             }
 90         }
 91
 92         /* End of critical section. */
 93         if (pthread_mutex_unlock(&mutex) != 0) {
 94             perror(progname);
 95             exit(12);
 96         }
 97     }
 98
 99     for (;;) {
100         /* lots of other work */
101     }
102 }
```

```
count has reached 10.
```


Use pthread_cond_timedwait instead of pthread_cond_wait in line 48

```
 1 /* Excerpt from /usr/include/sys/timers.h. */
 2
 3 typedef struct timespec {
 4     time_t   tv_sec;     /* seconds */
 5     long     tv_nsec;    /* nanoseconds */
 6 } timespec_t;
```

```
 1 #include <errno.h>
 2 #include <pthread.h>    /* don't need to include timers.h */
 3
 4     timespec_t how_long = {1, 0};
 5     timespec_t absolute;
 6
 7     if (pthread_get_expiration_np(&how_long, &absolute) != 0) {
 8         perror(progname);
 9         return 1;
10     }
11
12     printf ("%d seconds, %ld nanoseconds\n", absolute.tv_sec, absolute.tv_nsec);
13
14     if (pthread_cond_timedwait(&condition, &mutex, &absolute) != 0) {
15         if (errno == EAGAIN) {
16             printf ("The time has expired.\n");
17         } else {
18             perror(progname);
19             return 2;
20         }
21     }
```

```
1072224000 seconds, 809168000 nanoseconds
```

```
1$ bc
(2004 - 1970) * 60 * 60 * 24 * 365
1072224000
control-d
2$
```

Use pthread_cond_broadcast instead of pthread_cond_signal in line 86

`pthread_cond_signal` wakes only one waiting thread; **`pthread_cond_broadcast`** wakes them all (in order of scheduling priority).

```
1     if (pthread_cond_broadcast(&condition) != 0) {
2         perror(progname);
3         exit(11);
4     }
```

A spectacular example from the O'Reilly pthreads book, pp. 84−89

Assume that many threads want to read and write the same block of shared memory. It's okay for any number of threads to read the block simultaneously. But while one thread is writing, all other threads should be prevented from writing or reading.

There are two kinds of locks: a read lock and a write lock. A thread that tries to get a read lock will have to wait until there are no other threads writing. A thread that tries to get a write lock will have to wait until there are no other threads reading or writing.

Here's the user code:

```
 1 /* Initialization. */
 2 #include <pthread.h>
 3 pthread_rdwr_t lock;
 4
 5     if (pthread_rdwr_init_np(&lock, pthread_rdwr_default) != 0) {
 6         perror(progname);
 7         exit(1);
 8     }
```

```
 1     if (pthread_rdwr_rlock_np(&lock) != 0) {
 2         perror(progname);
 3         exit(2);
 4     }
 5
 6     /* Read the block of shared memory. */
 7
 8     if (pthread_rdwr_runlock_np(&lock) != 0) {
 9         perror(progname);
10         exit(3);
11     }
```

```
 1     if (pthread_rdwr_wlock_np(&lock) != 0) {
 2         perror(progname);
 3         exit(4);
 4     }
 5
 6     /* Write the block of shared memory. */
 7
 8     if (pthread_rdwr_wunlock_np(&lock) != 0) {
 9         perror(progname);
10         exit(5);
11     }
```

The implementation

Here's the catch: the data type **`pthread_rdwr_t`**, the value **`pthread_rdrw_default`**, and the five functions

```
    pthread_rdwr_init_np
    pthread_rdwr_rlock_np              pthread_rdwr_wlock_np
    pthread_rdwr_runlock_np            pthread_rdwr_wunlock_np
```

don't exist: we have to write them ourselves. For simplicity, we just **`return -1`** when anything goes wrong without trying to unlock the locks.

```
 1 typedef struct {
 2     int readers;
 3     int writers;
 4     pthread_mutex_t mutex;          /* protects readers and writers */
```



```
 5     pthread_cond_t condition;       /* readers == 0 || writers == 0 */
 6 } pthread_rdwr_t;
 7
 8 typedef void *pthread_rdwrattr_t;
 9 #define pthread_rdwr_default ((pthread_rdwrarttr_t)NULL)
10
11 int pthread_rdwr_init_np(pthread_rdwr_t *lock, pthread_rdwrattr_t attr)
12 {
13     lock->readers = 0;
14     lock->writers = 0;
15
16     if (pthread_mutex_init(&lock->mutex, pthread_mutexattr_default) != 0) {
17         return -1;
18     }
19
20     if (pthread_cond_init(&lock->condition, pthread_condattr_default) != 0) {
21         return -1;
22     }
23
24     return 0;
25 }
26
27 int pthread_rdwr_rlock_np(pthread_rdwr_t *lock)
28 {
29     if (pthread_mutex_lock(&lock->mutex) != 0) {
30         return -1;
31     }
32     /* Start of critical section. */
33
34     while (lock->writers > 0) {
35         if (pthread_cond_wait(&lock->condition, &lock->mutex) != 0) {
36             return -1;
37         }
38     }
39
40     ++lock->readers;
41
42     /* End of critical section. */
43     if (pthread_mutex_unlock(&lock->mutex) != 0) {
44         return -1;
45     }
46
47     return 0;
48 }
49
50 int pthread_rdwr_wlock_np(pthread_rdwr_t *lock)
51 {
52     if (pthread_mutex_lock(&lock->mutex) != 0) {
53         return -1;
54     }
55     /* Start of critical section. */
56
57     while (lock->readers > 0 || lock->writers > 0) {
58         if (pthread_cond_wait(&lock->condition, &lock->mutex) != 0) {
```



```
 59              return -1;
 60          }
 61      }
 62
 63      lock->writers = 1;
 64
 65      /* End of critical section. */
 66      if (pthread_mutex_unlock(&lock->mutex) != 0) {
 67          return -1;
 68      }
 69
 70      return 0;
 71  }
 72
 73  /*
 74  Only writers, not readers, could be waiting for the following signal, and only
 75  one writer could take advantage of it.  Therefore we call pthread_cond_signal
 76  instead of pthread_cond_broadcast.
 77  */
 78
 79  int pthread_rdwr_runlock_np(pthread_rdwr_t *lock)
 80  {
 81      if (pthread_mutex_lock(&lock->mutex) != 0) {
 82          return -1;
 83      }
 84      /* Start of critical section. */
 85
 86      if (lock->readers <= 0) {
 87          /* User forgot previous call to pthread_rdwr_rlock_np. */
 88          return -1;
 89      }
 90
 91      if (--lock->readers == 0) {
 92          if (pthread_cond_signal(&lock->condition, &lock->mutex) != 0) {
 93              return -1;
 94          }
 95      }
 96
 97      /* End of critical section. */
 98      if (pthread_mutex_unlock(&lock->mutex) != 0) {
 99          return -1;
100      }
101
102      return 0;
103  }
104
105  /*
106  Both writers and readers could be waiting for the following signal, and more
107  than one of them (i.e, all the readers) could take advantage of it.  Therefore
108  we call pthread_cond_broadcast instead of pthread_cond_signal.
109  */
110
111  int pthread_rdwr_wunlock_np(pthread_rdwr_t *lock)
112  {
```



```
113     if (pthread_mutex_lock(&lock->mutex) != 0) {
114         return -1;
115     }
116     /* Start of critical section. */
117
118     if (lock->writers != 1) {
119         /* User forgot previous call to pthread_rdwr_rwock_np. */
120         return -1;
121     }
122
123     lock->writers = 0;
124     if (pthread_cond_broadcast(&lock->condition, &lock->mutex) != 0) {
125         return -1;
126     }
127
128     /* End of critical section. */
129     if (pthread_mutex_unlock(&lock->mutex) != 0) {
130         return -1;
131     }
132
133     return 0;
134 }
```

Set the thread's stack size

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <pthread.h>
 4
 5 void *thread_function(void *p);
 6
 7 int main(int argc, char **argv)
 8 {
 9     pthread_t thread;
10     pthread_attr_t attr;
11     long size;
12
13     if (pthread_attr_create(&attr) != 0) {
14         perror(argv[0]);
15         return 1;
16     }
17
18     if (pthread_attr_setstacksize(&attr, 1024L * 10L) != 0) {
19         perror(argv[0]);
20         return 2;
21     }
22
23     size = pthread_attr_getstacksize(attr);
24     if (size < 0) {
25         perror(argv[0]);
26         return 3;
27     }
28     printf("The stack size is %ld.\n", size);
29
```

```
30      if (pthread_create(&thread, attr,
31          (pthread_startroutine_t)thread_function, NULL) != 0) {
32          perror(argv[0]);
33          return 4;
34      }
35
36      if (pthread_attr_delete(&attr) != 0) {
37          perror(argv[0]);
38          return 5;
39      }
40
41      if (pthread_detach(&thread) != 0) {
42          perror(argv[0]);
43          return 6;
44      }
45
46      return EXIT_SUCCESS;
47  }
48
49  void *thread_function(void *p)
50  {
51  }
```

```
The stack size is 10240.
```

Make sure that an initialization is performed exactly once

```
1  void init(void);
2
3  /* not initialized by calling a function: */
4  pthread_once_t once = pthread_once_init;
```

Each thread should do the following. It's more efficient than using a flag protected by a **mutex**.

```
1      if (pthread_once(&once, init) != 0) {
2          perror(progname);
3          exit(1);
4      }
5      /* At this point, init has been called exactly once. */
```

Give each thread its own data

```
1  typedef struct {
2      pthread_t thread;
3      int i;
4  } data;
5
6  #define N 3
7  data d[N];
8
9      int i;
10     for (i = 0; i < N; ++i) {
11         d[i].i = i;
12         if (pthread_create(&d[i].thread, pthread_attr_default,
13             (pthread_startroutine_t)thread_function, NULL) != 0) {
```



```
14              perror(progname);
15              exit(1);
16          }
17      }
```

Now each thread could say

```
 1      int i;
 2      thread_t thread;
 3
 4      for (i = 0; i < N; ++i) {
 5          thread = pthread_self();
 6          if (pthread_equal(&d[i].thread, &thread)) {
 7              goto found;
 8          }
 9      }
10
11      fprintf(stderr, "%s: couldn't find myself\n", progname);
12      exit(1);
13
14      found:;
15      Use d[i].i;
```

Give each thread its own data, using keys

```
 1  pthread_key_t key;
 2
 3      int i;
 4      pthread_t thread;
 5      char *p;
 6
 7      if (pthread_key_create(&key, free) != 0) {
 8          perror(progname);
 9          exit(1);
10      }
11
12      for (i = 0; i < N; ++i) {
13          if (pthread_create(&thread, pthread_attr_default,
14              (pthread_startroutine_t)thread_function, NULL) != 0) {
15              perror(progname);
16              exit(2);
17          }
18
19          p = malloc(10);
20          if (p == NULL) {
21              fprintf("%s: couldn't allocate memory\n", progname);
22              exit(3);
23          }
24
25          if (pthread_setspecific(key, (pthread_addr_t)p) != 0) {
26              perror(progname);
27              exit(4);
28          }
29      }
```


Now each thread could say

```
1     char *p;
2
3     if (pthread_getspecific(key, (pthread_addr_t)&p) != 0) {
4         perror(progname);
5         exit(4);
6     }
7
8     printf ("%s\n", p);
```

□

