

Fall 2006 Handout 9

Pthreads

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

<i>Unix system calls</i>	<i>Pthreads</i>
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

—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 *p);
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 labinfu.unipv.it: */
35     retval = inet_pton(AF_INET, "193.204.35.58", &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 labinfo.unipv.it.\n");
56
57     if (pthread_create(&thread, NULL, thread_function, (void *)&s) != 0) {
58         perror(progname);
59         return 6;
60     }
61
62     retval = pthread_detach(thread);
63     if (retval != 0) {
64         fprintf(stderr, "%s: pthread_detach returned %d.\n", argv[0], retval);
65         return 7;
66     }
67
68     sprintf(buffer, "ps -Lf -p %d", getpid()); /* -L "lightweight process" */
69     system(buffer);
70
71     /*
72     The main thread will copy from the server to the stdout.
73     */
74     while (read(s, &c, 1) == 1) {
75         putchar(c);
76     }
77
78     /*
79     Arrive here after receiving EOF from the server.
80     */
81     fprintf(stderr, "Connection closed by foreign host.\n");
82
83     /*
84     If pthread_cancel returns ESRCH, it means that the thread has already
85     died before we had a chance to cancel it.
86     */
87     retval = pthread_cancel(thread);
88     if (retval != 0 && retval != ESRCH) {
89         fprintf(stderr, "%s: pthread_cancel returned %d.\n", argv[0], retval);
90         return 8;
91     }
92
93     if (shutdown(s, SHUT_RDWR) != 0) {
94         perror(progname);
95         return 9;
96     }
97
98     return EXIT_SUCCESS;
99 }
100
101 void *thread_function(void *p)
102 {
103     const int fd = *(int *)p;
104     int i;
105
106     /*
107     This thread will copy from the stdin to the server.
108     */

```

```

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

```

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

```
4$ ls -l ~/bin/pthread1
```

LWP is “lightweight process”; NLWP is “number of lightweight processes”.

```
5$ pthread1
```

```
Trying 193.204.35.58...
```

```
Connected to labinfu.unipv.it.
```

UID	PID	PPID	LWP	NLWP	C	STIME	TTY	LTIME	CMD
mm64	28313	28311	1	2	0	09:41:17	pts/5	0:00	pthread1
mm64	28313	28311	2	2	0	09:41:17	pts/5	0:00	pthread1

```
"And this also," said Marlow suddenly,
```

```
"And this also," said Marlow suddenly,
```

```
"has been one of the dark places of the earth."
```

```
"has been one of the dark places of the earth."
```

```
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. I’m sorry the `free` is so far from the corresponding `malloc`. To move the `free` into the main thread, the main thread would have to call `pthread_join` instead of `pthread_detach`, and the child thread would have to call `pthread_exit` to return the the address of the allocated block.

—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 *p);
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 *p)
77 {
78     const int client = *(int *)p;
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`.

—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`.

—On the Web at

<http://i5.nyu.edu/~mm64/x52.9544/src/happens.c>

```

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

```

```

71      /* Start of critical section. */
72
73      ++count;
74
75      /* End of critical section. */
76      if (pthread_mutex_unlock(&mutex) != 0) {
77          perror(progname);
78          exit(8);
79      }
80  }
81
82  for (;;) {
83      /* lots of other work */
84  }
85 }

```

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
```

—On the Web at

<http://i5.nyu.edu/~mm64/x52.9544/src/condition.c>

```
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, NULL) != 0) {
20         perror(progname);
21         return 1;
22     }
23
24     if (pthread_cond_init(&condition, NULL) != 0) {
25         perror(progname);
26         return 2;
27     }
28
29     if (pthread_create(&thread, NULL, thread_function, NULL) != 0) {
30         perror(progname);
31         return 3;
32     }
33
34     if (pthread_detach(thread) != 0) {
35         perror(progname);
36         return 4;
37     }
38
39     /* Wait until count == n. */
40     if (pthread_mutex_lock(&mutex) != 0) {
41         perror(progname);
42         return 5;
43     }
44     /* Start of critical section. */
```

```
45
46     if (count != n) {                               /* change "if" to "while" */
47         if (pthread_cond_wait(&condition, &mutex) != 0) {
48             perror(progname);
49             return 6;
50         }
51     }
52
53     /* End of critical section. */
54     if (pthread_mutex_unlock(&mutex) != 0) {
55         perror(progname);
56         return 7;
57     }
58
59     if (pthread_cond_destroy(&condition) != 0) {
60         perror(progname);
61         return 8;
62     }
63
64     if (pthread_mutex_destroy(&mutex) != 0) {
65         perror(progname);
66         return 9;
67     }
68
69     printf ("count has reached %d.\n", n);
70     return EXIT_SUCCESS;
71 }
72
73 void *thread_function(void *p)
74 {
75     int i;
76
77     for (i = 0; i < n; ++i) {
78         if (pthread_mutex_lock(&mutex) != 0) {
79             perror(progname);
80             exit(10);
81         }
82         /* Start of critical section. */
83
84         if (++count == n) {
85             if (pthread_cond_signal(&condition) != 0) {
86                 perror(progname);
87                 exit(11);
88             }
89         }
90
91         /* End of critical section. */
92         if (pthread_mutex_unlock(&mutex) != 0) {
93             perror(progname);
94             exit(12);
95         }
96     }
97
98     for (;;) {
```

```

99         /* lots of other work */
100     }
101 }

```

```
count has reached 10.
```

Use `pthread_cond_timedwait` instead of `pthread_cond_wait` in the above line 47

```

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;
7
8
9 #include <errno.h>
10 #include <pthread.h> /* don't need to include timers.h */
11
12 timespec_t how_long = {1, 0};
13 timespec_t absolute;
14
15 if (pthread_get_expiration_np(&how_long, &absolute) != 0) {
16     perror(progname);
17     return 1;
18 }
19
20 printf ("%d seconds, %ld nanoseconds\n", absolute.tv_sec, absolute.tv_nsec);
21
22 if (pthread_cond_timedwait(&condition, &mutex, &absolute) != 0) {
23     if (errno == EAGAIN) {
24         printf ("The time has expired.\n");
25     } else {
26         perror(progname);
27         return 2;
28     }
29 }
30 }

```

```
1135296000 seconds, 809168000 nanoseconds
```

```

1$ bc
(2006 - 1970) * 60 * 60 * 24 * 365
1135296000
control-d
2$

```

Use `pthread_cond_broadcast` instead of `pthread_cond_signal` in the above line 85

`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_rwlock_t lock;
4
5     if (pthread_rwlock_init_np(&lock, pthread_rwlock_default) != 0) {
6         perror(progname);
7         exit(1);
8     }
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
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
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

The implementation

Here's the catch: the data type `pthread_rwlock_t`, the value `pthread_rwlock_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_rwlockattr_t;
9 #define pthread_rwlock_default ((pthread_rwlockattr_t)NULL)
10
11 int pthread_rwlock_init_np(pthread_rwlock_t *lock, pthread_rwlockattr_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_rwlock_rlock_np(pthread_rwlock_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_rwlock_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);

```

Expect

The standard book on Expect is

Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs
by Don Libes; O'Reilly & Associates, 1994; ISBN 1-56592-090-2

<http://www.oreilly.com/catalog/expect/>

<http://www.mel.nist.gov/msidstaff/libes/>

The Expect home page is <http://expect.nist.gov/>

Programs that require a dialog

ftp
telnet

nslookup
mail and the other mail readers
nn and the other news readers
dbx, **gdb**, and the other interactive debuggers
ksh requires the user to press control-z, control-c, etc.
vi and the other screen editors

passwd
fsck

Anonymous ftp

The Tcl home page <http://sunscript.sun.com/> tells where to get Tcl via anonymous **ftp**. Ominously, the **Name** prompt has a space after its final colon, but the **Password** prompt doesn't (Libes pp. 165–166).

```

1$ cd
2$ ftp ftp.sunlabs.com
Connected to rocky.sunlabs.com.
220 rocky FTP server (Version 4.125 Fri May 16 21:40:31 PDT 1997) ready.
Name (ftp.sunlabs.com:mm64): anonymous
331 Guest login ok, send your email address as password.
Password:mark.meretzky@nyu.edu
230- Guest login ok, access restrictions apply.
230- Local time is: Sat Feb 28 09:28:09 1998
230
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd pub/tcl
250 CWD command successful.
ftp> ls -l "| more"
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 307030
-rw-rw-r--  1 19663    51          8261 Jan 23 17:07 README
-rw-r--r--  1 19663   staff       11628 Nov 26 18:29 tcl8.0p2.patch.gz
-rw-r--r--  1 19663   staff    2449543 Nov 26 18:29 tcl8.0p2.tar.Z
-rw-r--r--  1 19663   staff    1522374 Nov 26 18:29 tcl8.0p2.tar.gz
226 Transfer complete.
ftp> binary
200 Type set to I.
ftp> get tcl8.0p2.patch.gz
200 PORT command successful.
150 Opening BINARY mode data connection for tcl8.0p2.patch.gz (11628 bytes).
226 Transfer complete.
11628 bytes received in 0.42 seconds (27 Kbytes/s)
ftp> quit
221 Goodbye.
3$

```

expect and send

The following script conducts the dialog with **telnet**. An **expect** command with an argument means “wait until I receive the following string”, or until **\$timeout** seconds have passed, or until end-of-file is received from the spawned process, whichever comes first. Therefore even if **Name** is never received, you would (eventually) still progress from line 8 to line 9. We’ll fix this later.

Thanks to the terminal driver, the argument of **send** must contain **\r** instead of the usual Unix **\n**. After all, the key that a human being actually hits is **RETURN**, not **LINEFEED** (Libes pp. 79–80). As in Tcl, the double quotes are unnecessary except when they enclose whitespace.

The **expect** command receives whatever output the child wrote to its **stdout**, **stderr**, or controlling **tty**. After all, all three of these would normally appear on the user’s screen. The **send** command sends input to the child which the child can read from either its **stdin** or controlling **tty**.

See Libes pp. 217–218 for the **--** on line 1.

```

1$ which expect
/opt/sfw/bin/expect

```

```

1 #!/opt/sfw/bin/expect --
2 #Get the file tcl8.0p2.patch.gz via anonymous ftp.
3
4 cd
5 set timeout 60                ;#default 10 seconds; -1 wait forever; 0 no wait
6 spawn ftp ftp.sunlabs.com
7
8 expect "Name"                 ;#no colon
9 send "anonymous\r"           ;#carriage return
10
11 expect "Password:"           ;#no space after colon
12 send "mark.meretzky@nyu.edu\r"
13
14 expect "ftp> "
15 send "cd pub/tcl\r"
16
17 expect "ftp> "
18 send "binary\r"
19
20 expect "ftp> "
21 send "get tcl8.0p2.patch.gz\r"
22
23 expect "ftp> "
24 send "quit\r"
25
26 exit 0

```

The interact command: Libes pp. 8–9, 82–83

The `interact` command lets the user take over:

```

1 #!/opt/sfw/bin/expect --
2
3 cd
4 set timeout 60
5 spawn ftp ftp.sunlabs.com
6
7 expect "Name"
8 send "anonymous\r"
9
10 expect "Password:"
11 send "mark.meretzky@nyu.edu\r"
12
13 expect "ftp> "
14 send "cd pub/tcl\r"
15
16 expect "ftp> "
17 send "ls -l \"| more \"\r"
18
19 interact                      ;#Put terminal in raw mode, Libes pp. 324, 344.

```

To save the user from having to watch the whole dialog, surround part of the program with a pair of `log_user` commands. The `send_user` command is the same as the Tcl command `puts`, except that `puts` automatically appends a `\n`. (And `send_error` is like `puts stderr`, Libes pp. 187–188).

```

1 #!/opt/sfw/bin/expect --
2
3 cd
4 send_user "Connecting to ftp.sunlabs.com...\n"      ;#newline
5 log_user 0
6 spawn ftp ftp.sunlabs.com
7
8 expect "Name"
9 send "anonymous\r"
10
11 expect "Password:"
12 send "mark.meretzky@nyu.edu\r"
13
14 expect "ftp> "
15 send "cd pub/tcl\r"
16 log_user 1
17
18 interact

```

Carriage return vs. newline: Libes pp. 72, 78–79

```

send_user "hello\n"      ;#just like printf("hello\n"); in a C program
expect_user "hello\n"

send "hello\r"          ;#makes the spawned process receive hello\n
expect "hello\r\n"      ;#from a spawned process that sent hello\n

```

Verify the above send.

The **expect** command with no arguments (Libes pp. 98, 104) reads all of the child's standard output and standard error output and copies it to the standard output of the Expect script. The command terminates when **eof** is encountered or when **\$timeout** seconds have elapsed, whichever comes first. Without the **expect** command, you would see nothing.

```

1 #!/opt/sfw/bin/expect --
2
3 spawn od -Ad -cvw1      ;#octal dump the characters, addresses in decimal
4 send "hello\r"
5 expect                  ;#and wait 10 seconds for timeout
6 exit 0

```

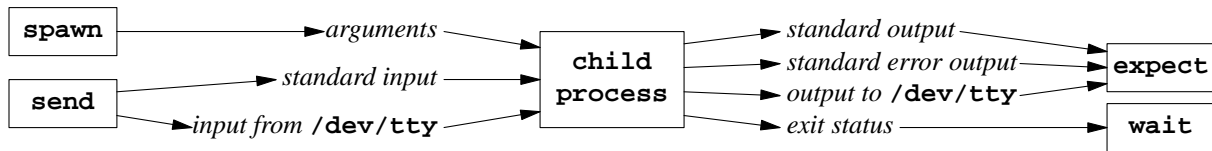
The output is

```

spawn od -Ad -cvw1
hello
0000000    h
0000001    e
0000002    l
0000003    l
0000004    o
0000005    \n

```

The I/O redirections performed by Expect: Libes pp. 174–175, 210–211, 291–293



The **spawn** command connects the child process's standard input, standard output, and standard error to a pseudoterminal. To change these defaults, spawn a shell that will overlay itself with the shell **exec** command (Libes pp. 291–293):

```
spawn /bin/sh -c "exec cal 13 1998 2> errorfile"
```

The data directed from a child process into the **expect** command is copied to the standard output of the Expect script, unless you say **log_user 0**.

Please email me if you can run ftp interactively from a perlscript.

Modify the following example so that it drives **ftp** instead of **bc**. Does **ftp** write to its standard output or its **/dev/tty** (or both)? Does **ftp** read from its standard input or its **/dev/tty** (or both)? Will **ftp** flush its output buffer when writing to a pipe, or will it flush only when writing output to a terminal? Can the perlscript direct **ftp**'s output to a pseudoterminal? Does the Perl standard library contain subroutines to do all this? For a discussion of the problems surrounding **Open2** in Perl, see pp. 344–345, 455–457 in the O'Reilly *Programming Perl, 2nd ed.*

Can the perlscript read the **ftp** prompts with the **<>** operator? Do you have to change the input record separator variable **\$/?** Or should you read the prompts with the **getc** function?

```

1 #!/bin/perl
2 use IPC::Open2;
3 use FileHandle;
4
5 $pid = open2(\*IN, \*OUT, 'bc');
6 defined $pid || die "$0: $!";
7 autoflush OUT 1;          #flush automatically after each print OUT
8
9 print OUT "1+2\n";
10 $line = <IN>;
11 print $line;
12
13 close IN;
14 close OUT;
15 exit 0;
  
```

The standard output of the perlscript is

```
3
```

```

1 #!/opt/sfw/bin/expect --
2
3 spawn bc
4 send "1+2\r"
5
6 expect "\n"          ;#As soon as we receive one line of standard output from bc,
7 close                ;#send an eof to bc.
8 exit 0
  
```


A process can tell the difference

A Unix process can tell if its standard output has been directed to a terminal:

```

    if (isatty(1)) {           /* C & C++; and #include <unistd.h> */
    if [ -t 1 ]; then         #Bourne shell
    if [[ -t 1 ]]; then       #Korn shell
    if (-t 1) {               #Perl
    if {$tcl_interactive} {   ;#Tcl, Tk, Expect

```

A child spawned by **exec** believes that its standard output has not been redirected to a terminal.

```

1 #!/usr/local/bin/tclsh
2 #Make ls believe its standard output is not directed to a terminal.
3
4 puts [exec ls]
5 exit 0

```

```

curly                                     single-column output
larry
moe

```

But a child spawned by **spawn** believes that its standard output has been redirected to a terminal.

```

1 #!/opt/sfw/bin/expect --
2 #Make ls believe its standard output is directed to a terminal.
3 spawn ls
4
5 expect
6 exit 0

```

```

curly      larry      moe                                     multi-column output

```

Deliberately bug-prone example

What will this C program print?

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 main()
5 {
6     printf ("hello\n");
7     _exit(0);           /* underscore exit */
8 }

```

```
1$ cc prog.c
```

```
2$ a.out
```

*Direct a.out's standard output to screen; it outputs **hello**.*

```
3$ a.out | cat
```

Direct a.out's standard output to pipe; it outputs nothing.

Expect with a pattern-action pair: Libes p. 75

Let's verify the above claim that a process that outputs **hello\n** will cause an **expect** command to receive **hello\r\n**:

```

1 #!/opt/sfw/bin/expect --
2
3 spawn echo hello
4
5 expect "hello\r\n"
6 send_user "I received hello CR LF.\n"
7 exit 0

```

Even if the pattern `hello\r\n` is never seen, the script will still print the message. That's because an `expect` command with a pattern will terminate when the pattern is first seen, or when `eof` is encountered, or when `$timeout` seconds have elapsed, whichever comes first.

To print the message only if the pattern is seen, give `expect` a pattern-action pair:

```

1 #!/opt/sfw/bin/expect --
2
3 spawn echo hello
4
5 expect "hello\r\n" {send_user "I received hello CR LF.\n"}
6 exit 0

```

Syntax for multiple pattern-action pairs: Libes pp. 75–77, 158–160

(1) You're allowed to omit the action of the last pattern-action pair. Therefore it's conventional to list the pattern with the biggest action last, and the patterns with the smallest actions (often error conditions) first. Other than that, it doesn't matter what order you list them in.

```

1 #Fragment of an expect script that spawns ftp.
2
3 expect "ftp> "
4 send "cd $dirname\r"
5
6 expect full_buffer          {exit 5} \
7     timeout                 {exit 4} \
8     eof                     {exit 4} \
9     "No such file or directory.\r\n" {exit 3} \
10    "Not a directory.\r\n"      {exit 2} \
11    "Permission denied.\r\n"   {exit 1} \
12    "CWD command successful.\r\n"
13
14 #Action for "CWD command was successful.\r\n" goes
15 #here.

```

(2) The pattern `default` means `timeout` or `eof`, whichever happens first (Libes p. 101):

```

1 expect "ftp> "
2 send "cd $dirname\r"
3
4 expect full_buffer {exit 5} \
5     default {exit 4} \
6     "No such file or directory.\r\n" {exit 3} \
7     "Not a directory.\r\n" {exit 2} \
8     "Permission denied.\r\n" {exit 1} \
9     "CWD command successful.\r\n"
10
11 #Action for "CWD command was successful.\r\n" goes
12 #here.

```

(3) Eliminate the backslashes as in the Tcl `if-elseif` command:

```

1 expect full_buffer {
2     exit 5
3 } default {
4     exit 4
5 } "No such file or directory.\r\n" {
6     exit 3
7 } "Not a directory.\r\n" {
8     exit 2
9 } "Permission denied.\r\n" {
10    exit 1
11 } "CWD command successful.\r\n"
12
13 #Action for "CWD command was successful.\r\n" goes
14 #here.

```

(4) You can also give the `expect` command exactly one argument, of a list of the above arguments. I simply surrounded them with a pair of curly braces:

```
1 expect {
2     full_buffer {
3         exit 5
4     }
5
6     default {
7         exit 4
8     }
9
10    "No such file or directory.\r\n" {
11        exit 3
12    }
13
14    "Not a directory.\r\n" {
15        exit 2
16    }
17
18    "Permission denied.\r\n" {
19        exit 1
20    }
21
22    "CWD command successful.\r\n"
23 }
24
25 #Action for "CWD command was successful.\r\n" goes
26 #here.
```

expect_before and expect_after: Libes pp. 101, 259–268

A conscientious programmer should write a pattern for `eof`, `timeout`, and `full_buffer` in every `expect` command. You can avoid this repetition with the `expect_before` and `expect_after` commands.

Control structure in Expect

`1$` is my shell prompt, and `?` is my `mail` prompt:

```

2$ mailx
Mail $Revision: 4.2.4.2 $ Type ? for help.
"/usr/spool/mail/mm64": 4 messages 3 unread
  1 abc1234 Sun May 10 10:30 13/293 "Dodsworth"
>U 2 def5678 Sun May 10 11:30 12/290 "help"
  U 3 ghi9012 Sun May 10 11:45 12/288 "Olaf Stapledon"
  U 4 abc1234 Sun May 10 11:46 12/289 "help"
? R2
To: def5678
Subject: Re: help

~r /home1/m/mm64/helpfile
"/home1/m/mm64/helpfile" 1/29
.
EOT
? d2
? R4
To: abc1234
Subject: Re: help

~r /home1/m/mm64/helpfile
"/home1/m/mm64/helpfile" 1/29
.
EOT
? d4
? q
3$

```

Since the `mail` prompt `?` is a special character to the `expect` commands in lines 19, 52, and 55, it must be preceded by a `\`. And since the `\` is a special character in Tcl, it too must be preceded by a `\`. See Libes, pp. 73, 121, for the `$` anchor when looking for a prompt.

The array element `$expect_out(buffer)` in line 24 contains all the characters that were received by the most recent successful `expect`—one line of text, ending with `\r\n`.

At line 35, the variable `$letters` contains a list of four elements, each of which is a list of two elements:

```
{1 Dodsworth} {2 help} {3 {Olaf Stapledon}} {4 help}
```

```

1 #!/opt/sfw/bin/expect --
2 #Mail a helpfile to everyone whose subject was "help"
3
4 spawn mailx
5
6 expect {
7     "No mail for mm64\r\n" {
8         exit 0
9     }
10
11     "Type \\\? for help.\r\n"
12 }
13 expect "\r\n"                ;#4 messages, 3 unread
14
15 #Loop until we encounter the ? prompt.
16 set letters {}

```

```

17 while {1} {
18     expect {
19         "\\? $" {
20             break
21         }
22
23         "\r\n" {
24             set line $expect_out(buffer)
25             set llen [llength $line]
26
27             set subject [lindex $line [expr $llen - 1]]
28             set n       [lindex $line [expr $llen - 8]]
29
30             set letter [list $n $subject]
31             lappend letters $letter
32         }
33     }
34 }
35
36 foreach letter $letters {
37     set subject [lindex $letter 1]
38     if {[string compare $subject "help"] == 0} {
39         set n [lindex $letter 0]
40
41         send "R$n\r"                ;#Reply
42         expect "\r\n"
43         expect "\r\n"
44         expect "\r\n"
45
46         send "~r /home/m/mm64/helpfile\r" ;#read file
47         expect "\r\n"
48
49         send ".\r"
50         expect "EOT\r\n"
51
52         expect "\\? $"
53         send "d$n\r"                ;#delete letter
54
55         expect "\\? $"
56     }
57 }
58
59 send "q\r"                ;#quit from mail
60 exit 0

```

Drive nslookup with expect

—On the Web at

<http://i5.nyu.edu/~mm64/x52.9547/src/zone.ex>

```

1 #!/opt/sfw/bin/expect --
2 #Perform a DNS zone transfer.
3
4 if {$argc != 1} {
5     puts stderr "$argv0: requires one command line argument\n"

```

```

6     exit 1
7 }
8 set domain [lindex $argv 0]
9
10 send_user "Connecting to $domain...\n"
11 log_user 0
12 set timeout 10    ;#default 10 seconds; -1 wait forever; 0 no wait
13 spawn /usr/sbin/nslookup
14
15 expect "> $"
16 send "set type=any\r"
17
18 expect "> $"
19 send "$domain\r"
20
21 set nameservers {}
22
23 while {1} {
24     expect {
25         "> $" {
26             break
27         }
28
29         -re "$domain\[ \]+\nserver = ([^\r\n\+])\r\n" {
30             lappend nameservers $expect_out(1,string)
31         }
32     }
33 }
34
35 #Remove duplicates from the list of nameservers.
36
37 for {set i 0} {$i < [llength $nameservers] - 1} {incr i} {
38     for {set j [expr $i + 1]} {$j < [llength $nameservers]} {incr j} {
39         if {[lindex $nameservers $i] == [lindex $nameservers $j]} {
40             #Remove element number j from the list.
41             set nameservers [lreplace $nameservers $j $j]
42         }
43     }
44 }
45
46 send_user "Found the following [llength $nameservers] nameservers:\n"
47 foreach nameserver $nameservers {
48     send_user "\t$nameserver\n"
49 }
50 send_user "\n"
51
52 log_user 1
53
54 foreach nameserver $nameservers {
55
56     send "server $nameserver\r"
57
58     expect "> $"
59     send "ls $domain\r"

```

```

60
61     expect -re "\\*\\*\\* Can't list domain $domain: Unspecified error\r\n" {
62         expect "> $"
63         continue
64     }
65
66     break    ;#want only one zone transfer
67 }
68
69 send "exit\r"
70 exit 0

```

```

1$ zone.ex uni-heidelberg.de
Connecting to uni-heidelberg.de...
Found the following 4 nameservers:
    sun0.urz.uni-heidelberg.de
    sun1.urz.uni-heidelberg.de
    dns1.belwue.de
    dns3.belwue.de

Trying nameserver sun0.urz.uni-heidelberg.de...
Trying nameserver sun1.urz.uni-heidelberg.de...
Trying nameserver dns1.belwue.de...
[dns1.belwue.de]
$ORIGIN uni-heidelberg.de.
dc1.ad                20M IN A           129.206.15.144
dc2.ad                1H  IN A           129.206.15.208
dc3.ad                1H  IN A           129.206.15.145
etc.
>
2$

```

sun1.urz.uni-heidelberg.de must be a router: it has two IP addresses.

```

3$ nslookup sun1.urz.uni-heidelberg.de
Server:  CMCL2.NYU.EDU
Address: 128.122.253.92

Non-authoritative answer:
Name:    sun1.urz.uni-heidelberg.de
Addresses: 129.206.210.127, 129.206.100.127

```

▼ Homework 9.1: get a small zone transfer

Can you get a zone transfer from NYU?

```
1$ zone.ex nyu.edu
```

Can you find a DNS server that will send you a zone transfer? Can you get it with the above Expect script? Can you use an Expect script to get a zone transfer from the DNS server that we're about to set up?

▲
□