Fall 2006 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

—On the Web at
`http://i5.nyu.edu/˜mm64/x52.9544/src/pthread1.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>   /* for getpid */
#include <string.h>
#include <pthread.h>
#include <errno.h>     /* for ESRCH */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

char *progname;
void *thread_function(void *p);

int main(int argc, char **argv)
{
    const int s = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in address;
    char buffer[INET_ADDRSTRLEN]; /* for inet_ntop */
    pthread_t thread;
    int retval;                   /* instead of errno */
    char c;                       /* write requires a char: KR p. 170 */

    progname = argv[0];
    if (s < 0) {
        perror(progname);
        return 1;
    }

    bzero((char *)&address, sizeof address);
    address.sin_family = AF_INET;
    address.sin_port = htons(7);

    /* IP address for labinfu.unipv.it: */
    retval = inet_pton(AF_INET, "193.204.35.58", &address.sin_addr);
    if (retval == 0) {
        fprintf(stderr, "%s: bad dotted string to inet_pton\n", argv[0]);
        return 2;
    } else if (retval != 1) {
        perror(argv[0]);
        return 3;
    }

    if (inet_ntop(AF_INET, &address.sin_addr, buffer, sizeof buffer) == NULL) {
        perror(argv[0]);
        return 4;
    }
    printf("Trying %s...\n", buffer);

    if (connect(s, (struct sockaddr *)&address, sizeof address) != 0) {
        perror(progname);
        return 5;
    }

    printf("Connected to labinfu.unipv.it.\n");

    if (pthread_create(&thread, NULL, thread_function, (void *)&s) != 0) {
        perror(progname);
        return 6;
    }

    retval = pthread_detach(thread);
    if (retval != 0) {
        fprintf(stderr, "%s: pthread_detach returned %d.\n", argv[0], retval);
        return 7;
    }

    sprintf(buffer, "ps -Lf -p %d", getpid()); /* -L "lightweight process" */
    system(buffer);

    /*
    The main thread will copy from the server to the stdout.
    */
    while (read(s, &c, 1) == 1) {
        putchar(c);
    }

    /*
    Arrive here after receiving EOF from the server.
    */
    fprintf(stderr, "Connection closed by foreign host.\n");

    /*
    If pthread_cancel returns ESRCH, it means that the thread has already
    died before we had a chance to cancel it.
    */
    retval = pthread_cancel(thread);
    if (retval != 0 && retval != ESRCH) {
        fprintf(stderr, "%s: pthread_cancel returned %d.\n", argv[0], retval);
        return 8;
    }

    if (shutdown(s, SHUT_RDWR) != 0) {
        perror(progname);
        return 9;
    }

    return EXIT_SUCCESS;
}

void *thread_function(void *p)
{
    const int fd = *(int *)p;
    int i;

    /*
    This thread will copy from the stdin to the server.
    */
```

```
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;
```

```
void *thread_function(void *p);

int main(int argc, char **argv)
{
    const int s = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in myaddress;
    struct sockaddr_in clientaddress;
    socklen_t length = sizeof clientaddress;

    progname = argv[0];
    if (s < 0) {
        perror(progname);
        return 1;
    }

    bzero((char *)&myaddress, sizeof myaddress);
    myaddress.sin_family = AF_INET;
    myaddress.sin_port = htons(9266);
    myaddress.sin_addr.s_addr = INADDR_ANY;

    if (bind(s, (const struct sockaddr *)&myaddress, sizeof myaddress) != 0) {
        perror(progname);
        return 2;
    }

    if (listen(s, SOMAXCONN) != 0) {
        perror(progname);
        return 3;
    }

    for (;;) {
        pthread_t thread;
        int retval;

        /*
        file descriptor for talking to each new client
        */
        int *const pclient = malloc(sizeof (int));
        if (pclient == NULL) {
            perror(progname);
            return 4;
        }

        *pclient = accept(s, (struct sockaddr *)&clientaddress, &length);
        if (*pclient < 0) {
            perror(progname);
            return 5;
        }

        retval = pthread_create(&thread, NULL, thread_function, pclient);
        if (retval != 0) {
            fprintf(stderr, "%s: pthread_create returned %d.\n",
                argv[0], retval);
            return 6;
        }

        retval = pthread_detach(thread);
        if (retval) {
            fprintf(stderr, "%s: pthread_detach returned %d.\n",
                argv[0], retval);
            return 7;
        }
    }
}

void *thread_function(void *p)
{
    const int client = *(int *)p;
    char buffer[2];

    /*
    The service provided by this server is merely to input one character
    and then output it in lowercase, followed by a newline.
    */

    if (read(client, buffer, 1) != 1) {
        perror(progname);
        exit(8);
    }

    buffer[0] = tolower(buffer[0]);
    buffer[1] = '\n';

    if (write(client, buffer, sizeof buffer) != sizeof buffer) {
        perror(progname);
        exit(9);
    }

    if (shutdown(client, SHUT_RDWR) != 0) {
        perror(progname);
        exit(10);
    }

    free(s);
}
```

```
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`

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *p);

const char *progname;
int n = 0;
pthread_mutex_t mutex;   /* protects n */

int main(int argc, char **argv)
{
    pthread_t thread;
    void *status;
    int retval;

    progname = argv[0];

    retval = pthread_mutex_init(&mutex, NULL);
    if (retval != 0) {
        fprintf(stderr, "%s: pthread_mutex_init returned %d.\n",
            argv[0], retval);
        return 1;
    }

    if (pthread_create(&thread, NULL,
        thread_function, &n) != 0) {
        perror(progname);
        return 2;
    }

    if (pthread_mutex_lock(&mutex) != 0) {
        perror(progname);
        return 3;
    }
    /* Start of critical section. */

    ++n;

    /* End of critical section. */
    if (pthread_mutex_unlock(&mutex) != 0) {
        perror(progname);
        return 4;
    }

    if (pthread_join(thread, &status) != 0) {     /* like waitpid */
        perror(progname);
        return 5;
    }

    if (pthread_mutex_destroy(&mutex) < 0) {
        perror(progname);
        return 6;
    }

    printf("n == %d, and the thread returned \"%s\".\n",
        n, (char *)status);
}

void *thread_function(void *p)
{
    if (pthread_mutex_lock(&mutex) != 0) {
        perror(progname);
        exit(7);
    }
    /* Start of critical section. */

    ++*(int *)p;

    /* End of critical section. */
    if (pthread_mutex_unlock(&mutex) != 0) {
        perror(progname);
        exit(8);
    }

    pthread_exit("goodbye");
}
```

```
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`

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

const int n = 10;
int count = 0;
pthread_mutex_t mutex;   /* protects count */
char *progname;

void *thread_function(void *p);

int main(int argc, char **argv)
{
    pthread_t thread;
    int keep_looping = 1;

    progname = argv[0];

    if (pthread_mutex_init(&mutex, NULL) != 0) {
        perror(progname);
        return 1;
    }

    if (pthread_create(&thread, NULL, thread_function, NULL) != 0) {
        perror(progname);
        return 2;
    }

    if (pthread_detach(thread) != 0) {
        perror(progname);
        return 3;
    }

    /* Wait until count == n. */
    while (keep_looping) {
        if (pthread_mutex_lock(&mutex) != 0) {
            perror(progname);
            return 4;
        }
        /* Start of critical section. */

        if (count == n) {
            keep_looping = 0;
        }

        /* End of critical section. */
        if (pthread_mutex_unlock(&mutex) != 0) {
            perror(progname);
            return 5;
        }
    }

    if (pthread_mutex_destroy(&mutex) != 0) {
        perror(progname);
        return 6;
    }

    printf("count has reached %d.\n", n);
    return EXIT_SUCCESS;
}

void *thread_function(void *p)
{
    int i;

    for (i = 0; i < n; ++i) {
        if (pthread_mutex_lock(&mutex) != 0) {
            perror(progname);
            exit(7);
        }
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. */
```

```

    if (count != n) {                           /* change "if" to "while" */
        if (pthread_cond_wait(&condition, &mutex) != 0) {
            perror(progname);
            return 6;
        }
    }

    /* End of critical section. */
    if (pthread_mutex_unlock(&mutex) != 0) {
        perror(progname);
        return 7;
    }

    if (pthread_cond_destroy(&condition) != 0) {
        perror(progname);
        return 8;
    }

    if (pthread_mutex_destroy(&mutex) != 0) {
        perror(progname);
        return 9;
    }

    printf ("count has reached %d.\n", n);
    return EXIT_SUCCESS;
}

void *thread_function(void *p)
{
    int i;

    for (i = 0; i < n; ++i) {
        if (pthread_mutex_lock(&mutex) != 0) {
            perror(progname);
            exit(10);
        }
        /* Start of critical section. */

        if (++count == n) {
            if (pthread_cond_signal(&condition) != 0) {
                perror(progname);
                exit(11);
            }
        }

        /* End of critical section. */
        if (pthread_mutex_unlock(&mutex) != 0) {
            perror(progname);
            exit(12);
        }
    }

    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

```
/* Excerpt from /usr/include/sys/timers.h. */

typedef struct timespec {
    time_t  tv_sec;     /* seconds */
    long    tv_nsec;    /* nanoseconds */
} timespec_t;
```

```
#include <errno.h>
#include <pthread.h>    /* don’t need to include timers.h */

    timespec_t how_long = {1, 0};
    timespec_t absolute;

    if (pthread_get_expiration_np(&how_long, &absolute) != 0) {
        perror(progname);
        return 1;
    }

    printf ("%d seconds, %ld nanoseconds\n", absolute.tv_sec, absolute.tv_nsec);

    if (pthread_cond_timedwait(&condition, &mutex, &absolute) != 0) {
        if (errno == EAGAIN) {
            printf ("The time has expired.\n");
        } else {
            perror(progname);
            return 2;
        }
    }
```

```
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).

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

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:

```
/* Initialization. */
#include <pthread.h>
pthread_rdwr_t lock;

    if (pthread_rdwr_init_np(&lock, pthread_rdwr_default) != 0) {
        perror(progname);
        exit(1);
    }
```

```
    if (pthread_rdwr_rlock_np(&lock) != 0) {
        perror(progname);
        exit(2);
    }

    /* Read the block of shared memory. */

    if (pthread_rdwr_runlock_np(&lock) != 0) {
        perror(progname);
        exit(3);
    }
```

```
    if (pthread_rdwr_wlock_np(&lock) != 0) {
        perror(progname);
        exit(4);
    }

    /* Write the block of shared memory. */

    if (pthread_rdwr_wunlock_np(&lock) != 0) {
        perror(progname);
        exit(5);
    }
```

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.

```
typedef struct {
    int readers;
    int writers;
    pthread_mutex_t mutex;          /* protects readers and writers */
    pthread_cond_t condition;       /* readers == 0 || writers == 0 */
} pthread_rdwr_t;

typedef void *pthread_rdwrattr_t;
#define pthread_rdwr_default ((pthread_rdwrarttr_t)NULL)

int pthread_rdwr_init_np(pthread_rdwr_t *lock, pthread_rdwrattr_t attr)
{
    lock->readers = 0;
    lock->writers = 0;

    if (pthread_mutex_init(&lock->mutex, pthread_mutexattr_default) != 0) {
        return -1;
    }

    if (pthread_cond_init(&lock->condition, pthread_condattr_default) != 0) {
        return -1;
    }

    return 0;
}

int pthread_rdwr_rlock_np(pthread_rdwr_t *lock)
{
    if (pthread_mutex_lock(&lock->mutex) != 0) {
        return -1;
    }
    /* Start of critical section. */

    while (lock->writers > 0) {
        if (pthread_cond_wait(&lock->condition, &lock->mutex) != 0) {
            return -1;
        }
    }

    ++lock->readers;

    /* End of critical section. */
    if (pthread_mutex_unlock(&lock->mutex) != 0) {
        return -1;
    }

    return 0;
}
```

```

int pthread_rdwr_wlock_np(pthread_rdwr_t *lock)
{
    if (pthread_mutex_lock(&lock->mutex) != 0) {
        return -1;
    }
    /* Start of critical section. */

    while (lock->readers > 0 || lock->writers > 0) {
        if (pthread_cond_wait(&lock->condition, &lock->mutex) != 0) {
            return -1;
        }
    }

    lock->writers = 1;

    /* End of critical section. */
    if (pthread_mutex_unlock(&lock->mutex) != 0) {
        return -1;
    }

    return 0;
}

/*
Only writers, not readers, could be waiting for the following signal, and only
one writer could take advantage of it.  Therefore we call pthread_cond_signal
instead of pthread_cond_broadcast.
*/

int pthread_rdwr_runlock_np(pthread_rdwr_t *lock)
{
    if (pthread_mutex_lock(&lock->mutex) != 0) {
        return -1;
    }
    /* Start of critical section. */

    if (lock->readers <= 0) {
        /* User forgot previous call to pthread_rdwr_rlock_np. */
        return -1;
    }

    if (--lock->readers == 0) {
        if (pthread_cond_signal(&lock->condition, &lock->mutex) != 0) {
            return -1;
        }
    }

    /* End of critical section. */
    if (pthread_mutex_unlock(&lock->mutex) != 0) {
        return -1;
    }

    return 0;
}

/*
Both writers and readers could be waiting for the following signal, and more
than one of them (i.e, all the readers) could take advantage of it.  Therefore
we call pthread_cond_broadcast instead of pthread_cond_signal.
*/

int pthread_rdwr_wunlock_np(pthread_rdwr_t *lock)
{
    if (pthread_mutex_lock(&lock->mutex) != 0) {
        return -1;
    }
    /* Start of critical section. */

    if (lock->writers != 1) {
        /* User forgot previous call to pthread_rdwr_rwock_np. */
        return -1;
    }

    lock->writers = 0;
    if (pthread_cond_broadcast(&lock->condition, &lock->mutex) != 0) {
        return -1;
    }

    /* End of critical section. */
    if (pthread_mutex_unlock(&lock->mutex) != 0) {
        return -1;
    }

    return 0;
}
```

Set the thread's stack size

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *p);

int main(int argc, char **argv)
{
    pthread_t thread;
    pthread_attr_t attr;
    long size;

    if (pthread_attr_create(&attr) != 0) {
        perror(argv[0]);
        return 1;
    }

    if (pthread_attr_setstacksize(&attr, 1024L * 10L) != 0) {
        perror(argv[0]);
        return 2;
    }

    size = pthread_attr_getstacksize(attr);
    if (size < 0) {
        perror(argv[0]);
        return 3;
    }
    printf("The stack size is %ld.\n", size);

    if (pthread_create(&thread, attr,
        (pthread_startroutine_t)thread_function, NULL) != 0) {
        perror(argv[0]);
        return 4;
    }

    if (pthread_attr_delete(&attr) != 0) {
        perror(argv[0]);
        return 5;
    }

    if (pthread_detach(&thread) != 0) {
        perror(argv[0]);
        return 6;
    }

    return EXIT_SUCCESS;
}

void *thread_function(void *p)
{
}
```

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

Make sure that an initialization is performed exactly once

```
void init(void);

/* not initialized by calling a function: */
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`.

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

Give each thread its own data

```
typedef struct {
    pthread_t thread;
    int i;
} data;

#define N 3
data d[N];

    int i;
    for (i = 0; i < N; ++i) {
        d[i].i = i;
        if (pthread_create(&d[i].thread, pthread_attr_default,
            (pthread_startroutine_t)thread_function, NULL) != 0) {
            perror(progname);
            exit(1);
        }
    }
```

Now each thread could say

```
    int i;
    thread_t thread;

    for (i = 0; i < N; ++i) {
        thread = pthread_self();
        if (pthread_equal(&d[i].thread, &thread)) {
            goto found;
        }
    }

    fprintf(stderr, "%s: couldn’t find myself\n", progname);
    exit(1);

    found:;
    Use d[i].i;
```

Give each thread its own data, using keys

```
pthread_key_t key;

    int i;
    pthread_t thread;
    char *p;

    if (pthread_key_create(&key, free) != 0) {
        perror(progname);
        exit(1);
    }

    for (i = 0; i < N; ++i) {
        if (pthread_create(&thread, pthread_attr_default,
            (pthread_startroutine_t)thread_function, NULL) != 0) {
            perror(progname);
            exit(2);
        }

        p = malloc(10);
        if (p == NULL) {
            fprintf("%s: couldn’t allocate memory\n", progname);
            exit(3);
        }

        if (pthread_setspecific(key, (pthread_addr_t)p) != 0) {
            perror(progname);
            exit(4);
        }
    }
```

Now each thread could say

```
    char *p;

    if (pthread_getspecific(key, (pthread_addr_t)&p) != 0) {
        perror(progname);
        exit(4);
    }

    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
```

```
#!/opt/sfw/bin/expect --
#Get the file tcl8.0p2.patch.gz via anonymous ftp.

cd
set timeout 60              ;#default 10 seconds; -1 wait forever; 0 no wait
spawn ftp ftp.sunlabs.com

expect "Name"               ;#no colon
send "anonymous\r"          ;#carriage return

expect "Password:"          ;#no space after colon
send "mark.meretzky@nyu.edu\r"

expect "ftp> "
send "cd pub/tcl\r"

expect "ftp> "
send "binary\r"

expect "ftp> "
send "get tcl8.0p2.patch.gz\r"

expect "ftp> "
send "quit\r"

exit 0
```

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

The **`interact`** command lets the user take over:

```
#!/opt/sfw/bin/expect --

cd
set timeout 60
spawn ftp ftp.sunlabs.com

expect "Name"
send "anonymous\r"

expect "Password:"
send "mark.meretzky@nyu.edu\r"

expect "ftp> "
send "cd pub/tcl\r"

expect "ftp> "
send "ls -l \"| more \"\r"

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).

```
#!/opt/sfw/bin/expect --

cd
send_user "Connecting to ftp.sunlabs.com...\n"    ;#newline
log_user 0
spawn ftp ftp.sunlabs.com

expect "Name"
send "anonymous\r"

expect "Password:"
send "mark.meretzky@nyu.edu\r"

expect "ftp> "
send "cd pub/tcl\r"
log_user 1

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.

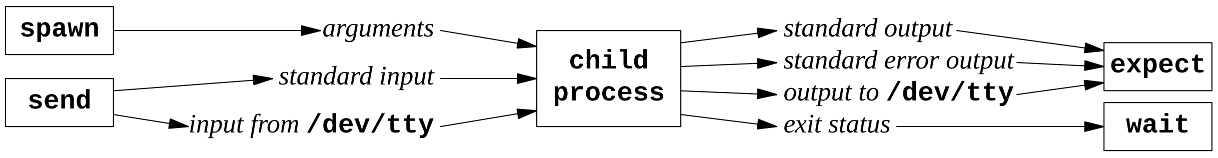
```
#!/opt/sfw/bin/expect --

spawn od -Ad -cvw1      ;#octal dump the characters, addresses in decimal
send "hello\r"
expect                  ;#and wait 10 seconds for timeout
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?

```
#!/bin/perl
use IPC::Open2;
use FileHandle;

$pid = open2(\*IN, \*OUT, 'bc');
defined $pid || die "$0: $!";
autoflush OUT 1;        #flush automatically after each print OUT

print OUT "1+2\n";
$line = <IN>;
print $line;

close IN;
close OUT;
exit 0;
```

The standard output of the perlscript is

```
3
```

```
#!/opt/sfw/bin/expect --

spawn bc
send "1+2\r"

expect "\n"             ;#As soon as we receive one line of standard output from bc,
close                   ;#send an eof to bc.
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.

```
#!/usr/local/bin/tclsh
#Make ls believe its standard output is not directed to a terminal.

puts [exec ls]
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.

```
#!/opt/sfw/bin/expect --
#Make ls believe its standard output is directed to a terminal.
spawn ls

expect
exit 0
```

```
curly     larry     moe                 multi-column output
```

Deliberately bug-prone example

What will this C program print?

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

main()
{
    printf ("hello\n");
    _exit(0);                       /* underscore exit */
}
```

```
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**:

```
#!/opt/sfw/bin/expect --

spawn echo hello

expect "hello\r\n"
send_user "I received hello CR LF.\n"
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:

```
#!/opt/sfw/bin/expect --

spawn echo hello

expect "hello\r\n" {send_user "I received hello CR LF.\n"}
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.

```
#Fragment of an expect script that spawns ftp.

expect "ftp> "
send "cd $dirname\r"

expect full_buffer                   {exit 5} \
    timeout                          {exit 4} \
    eof                              {exit 4} \
    "No such file or directory.\r\n" {exit 3} \
    "Not a directory.\r\n"           {exit 2} \
    "Permission denied.\r\n"         {exit 1} \
    "CWD command successful.\r\n"

#Action for "CWD command was successful.\r\n" goes
#here.
```

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

```
expect "ftp> "
send "cd $dirname\r"

expect full_buffer                     {exit 5} \
    default                            {exit 4} \
    "No such file or directory.\r\n" {exit 3} \
    "Not a directory.\r\n"           {exit 2} \
    "Permission denied.\r\n"         {exit 1} \
    "CWD command successful.\r\n"

#Action for "CWD command was successful.\r\n" goes
#here.
```

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

```
expect full_buffer {
        exit 5
    } default {
        exit 4
    } "No such file or directory.\r\n" {
        exit 3
    } "Not a directory.\r\n" {
        exit 2
    } "Permission denied.\r\n" {
        exit 1
    } "CWD command successful.\r\n"

#Action for "CWD command was successful.\r\n" goes
#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:

```
expect {
    full_buffer {
        exit 5
    }

    default {
        exit 4
    }

    "No such file or directory.\r\n" {
        exit 3
    }

    "Not a directory.\r\n" {
        exit 2
    }

    "Permission denied.\r\n" {
        exit 1
    }

    "CWD command successful.\r\n"
}

#Action for "CWD command was successful.\r\n" goes
#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 {}
```

```
while {1} {
    expect {
        "\\? $"  {
            break
        }

        "\r\n" {
            set line $expect_out(buffer)
            set llen [llength $line]

            set subject [lindex $line [expr $llen - 1]]
            set n       [lindex $line [expr $llen - 8]]

            set letter [list $n $subject]
            lappend letters $letter
        }
    }
}

foreach letter $letters {
    set subject [lindex $letter 1]
    if {[string compare $subject "help"] == 0} {
        set n [lindex $letter 0]

        send "R$n\r"                        ;#Reply
        expect "\r\n"
        expect "\r\n"
        expect "\r\n"

        send "~r /home/m/mm64/helpfile\r"   ;#read file
        expect "\r\n"

        send ".\r"
        expect "EOT\r\n"

        expect "\\? $"
        send "d$n\r"                        ;#delete letter

        expect "\\? $"
    }
}

send "q\r"                                  ;#quit from mail
exit 0
```

Drive nslookup with expect

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/zone.ex

```
#!/opt/sfw/bin/expect --
#Perform a DNS zone transfer.

if {$argc != 1} {
    puts stderr "$argv0: requires one command line argument\n"
    exit 1
}
set domain [lindex $argv 0]

send_user "Connecting to $domain...\n"
log_user 0
set timeout 10   ;#default 10 seconds; -1 wait forever; 0 no wait
spawn /usr/sbin/nslookup

expect "> $"
send "set type=any\r"

expect "> $"
send "$domain\r"

set nameservers {}

while {1} {
    expect {
        "> $" {
            break
        }

        -re "$domain\[   \]+nameserver = (\[^\r\n\]+)\r\n" {
            lappend nameservers $expect_out(1,string)
        }
    }
}

#Remove duplicates from the list of nameservers.

for {set i 0} {$i < [llength $nameservers] - 1} {incr i} {
    for {set j [expr $i + 1]} {$j < [llength $nameservers]} {incr j} {
        if {[lindex $nameservers $i] == [lindex $nameservers $j]} {
            #Remove element number j from the list.
            set nameservers [lreplace $nameservers $j $j]
        }
    }
}

send_user "Found the following [llength $nameservers] nameservers:\n"
foreach nameserver $nameservers {
    send_user "\t$nameserver\n"
}
send_user "\n"

log_user 1

foreach nameserver $nameservers {

    send "server $nameserver\r"

    expect "> $"
    send "ls $domain\r"

    expect -re "\\*\\*\\* Can’t list domain $domain: Unspecified error\r\n" {
        expect "> $"
        continue
    }

    break   ;#want only one zone transfer
}

send "exit\r"
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?
▲
□