Chapter 12
A Threaded Server

Motivation: Threads give you concurrency, without as much overhead as processes.

Paradigm: A thread is a procedure call that runs simultaneously with the parent.

Variables local to the procedure are unique to the thread.

Global variables are shared by all threads.
Common globals make it:
1) easy to exhange data
2) necessary to use mutual exclusion when accessing global data

The descriptor table is global (shared by all threads).

Downside: if any thread crashes, the process crashes taking all threads with it.

Note: if any thread uses `exit()`, the process exits taking all threads with it.

Creating A Thread

```
int pthread_create(
      pthread_t *thread,
      pthread_attr_t *attr,
      void *(*start_routine)(void *),
      void *arg);
```

Return value: zero=OK, non-zero=error

`thread`: the thread identifier (similar to a processID) is stored in the variable passed to this argument.

`attr`: special attributes the thread should have.

`start_routine`: the thread is this procedure. When the procedure returns, the thread is done/terminated. This procedure takes one argument (a pointer to something) and returns one argument (a pointer to something). Warning: type checking is often overridden for these values.

`arg`: This argument is to be passed to the thread procedure. `start_routine` will see this as its single incoming argument.

```
int pthread_attr_init(pthread_attr_t *attr);
```

Sets the all the attribute fields in the variable passed to their default values. Used to modify the attribute variable that will be used by the create.

```
int pthread_attr_setdetachstate(
      pthread_attr_t *attr,
      int detachstate);
```

The detach state attribute is either joinable, indicating the parent thread can do a `pthread_join` to wait for the child thread to exit. Or detached, indicating the join will not be done.

This call modifies the attribute variable. Do this before using the attribute varible to create the thread.

Warning: joinable ties up some thread resouces until the parent does do a join (recall the reaper problem). If you use this be sure your main thread does join on the exiting child threads.

A Threaded server is very similar to the concurrent server from Chapter 11.

This discussion will highlight the modification to the code from Chapter 11.

We will do this twice.

First time, the simple code.

We will do only echo with threads which will be very similar to the echo server from the previous chapter. This will be only the basic echo service, without the extra stats and mutex stuff that Comer adds..

Second time, Comer's TCP echo server code.

Comer's code adds some extra (unnecessary) things to the server. These items maintain statistics about what the server had done. Since all the threads place their information in the same set of global variables they need to use of semaphores to keep from having race conditions.

Simple Version

The includes:

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

Only the above include needs to be added. This is where the thread routines are defined.

```
int main(int argc, char *argv)
{
  pthread_t th;
  pthread_attr_t ta;
  char *service="echo";
  ...
  switch
  ...
  msock = passiveTCP(service, QLEN);
```

The only necessary differences are the at in this section of code is the declaration of the two thread variables.

`th` is the thread.
`ta` is the thread attributes.

```
msock = passiveTCP(service, QLEN);
(void) pthread_attr_init(&ta);
(void) pthread_attr_setdetachstate(&ta,
                PTHREAD_CREATE_DETACHED);
while (1)
...
```

The thread attribute variable is initialized to have defaults values.

The detach state is set to "detached" as opposed to "joinable".
The main program can't do thread joins.
The main program doesn't need to do thread joins to recover thread resources.

```
while (1)
  ...
  ssock = accept...
  if (ssock < 0) ...
  if (pthread_create(&th, &ta,
    (void * (*)(void *))TCPechod, (void *)ssock)<0)
     errexit("pthread_create:%s\n",strerror(errno));
}
```

Again, the server waits at the `accept` call until a client does a connect.
The switch/fork of the last chapter is replaced by a (simpler) call to create a thread.
Since threads do not get copies of the global variables (they share a single copy), we don't need to (must not) close the msock and sock.
The main program (parent) continues after the `pthead_create`, at the instruction after the if statement. (Goes back around to the accept.)
The new thread (child) continues with a procedure call to `TCPechod`. When it returns from this call the thread terminates.
If the create fails, error exit.

Create Details

```
pthread_create(&th, &ta,
  (void * (*)(void *))TCPechod, (void *)ssock)
```

`th` is filled in with information about the thread. This information is never used. (In the last chapter, the pid returned by fork wasn't really used.)

`ta` says to do a "detached" create. (It is not modified.)

`TCPechod` is the procedure that is to be the thread.

`(void * (*)(void *)`:
`TCPechod` is: `int (*)(int)`, it takes an integer and returns an integer.
Warning: this cast is drastically disabling type checking.

Meaning:
A function/subroutine is an address `(*)`
Has one parameter that is a pointer (`void *`)
Returns a pointer.

`(void *)ssock`: the socket to talk to the client will be passed as the parameter to the procedure when that procedure is created This allows the procedure to be the same as in the last chapter. Again, type checking is being disabled.

TCPechod

```
int TCPechod(int fd)
{
...
  close(fd);
}
```

Close the socket before the "return".

Summary

Add one include.
Add two thread related variables.
Add two calls to initialize the thread attributes variable.
Replace the `fork` switch with the `pthread_create` if.
Close the socket before the thread completes (returns).

Compiling With The pthread Library

Only the standard library is automatically included, others need to be explicitly named.
To explicitly name a library you use the `-l` option:

```
gcc myserver.c -lpthread -o server.out
```

Note: Comer adds the statistic tracking and reporting to the echo. These are extra, non-required items. We will discuss these in the remainder of this lecture.

Mutual exclusion

Comer enhanced code uses global variables to keep statics.

Because multiple thread are updating there variables, access to these variable needs to use critical sections and semaphores.

Because threads are in a single process they don't need to use the global (interprocess semaphores). They can use a less resource intensive set called thread semaphores.

The thread semaphore utilities are:

Declare a semaphore:

```
pthread_mutex_t semaphore;
```

Initialize a semaphore:

```
pthread_mutex_init(&semaphore);
```

Lock: entering critical section (`wait`):

```
pthread_mutex_lock(&semaphore);
```

Unlock: leaving critical section (`signal`)

```
pthread_mutex_unlock(&semaphore);
```

Enhanced echo
(Second Time)

We emphasize, just the differences from simple version.

```
#include <sys/resource.h>
```

This include is so we can track usage (resource) statistics.

```
struct {
 pthread_mutex_t st_mutex;
 unsigned int st_concount;
 unsigned int st_contotal;
 unsigned int st_contime;
 unsigned int st_bytecount;
} stats;
```

A global variable that keeps track of the statistics.

`prstats`: a procedure to print the statistics found in the global variable. This procedure is an infinite loop that does a print every 5 seconds.

```
(void) pthread_mutex_init(&stats.st_mutex,0);
```

Initialize the semaphore. This must be done before creating any threads.

```
if (pthread_create(&th,&ta,
    (void*(*)void*)prstats,0)<0)
      errexit...
```

Startup the thread (procedure that prints the statistics.

The remaining differences are in the procedures.

Side discussion: (the global variables)

`st_concount`: connection count. How many clients are connected right now. This is equal to the number of `TCPechod` procedures that are running.

`st_contotal`: connection total. How many clients have connected since the server started.

`st_conbytecount`: How many total bytes have been echoed since the server started.

`st_contime`: Total of all client connection times in seconds.

Additions to TCPechod procedure

```
time_t start;
...
start = time(0);
```

A local varible is used to compute the time we spend in this call to the procedure. Copies of the local variables exist in all threads so no locking is needed.

```
(void) pthread_mutex_lock(&stats.st_mutex);
stats.st_concount++;
(void) pthread_mutex_unlock(&stats.st_mutex);
```

Lock, change the global, unlock.

This code is done three separate places in this procedure.

prstats summary

lock, print stats, unlock, sleep 5 seconds.