CS61 Lecture Scribe Notes - Lecture 22

November 20, 2014

Transcribed and compiled by Group 2

Things being covered today:
-network programming
-synchronization
-threads

Looking at `service-server02.c`:
-Fragile server - by connecting to the server, we can prevent it from handling anyone else's connections (i.e., it's vulnerable to a *denial-of-service attack*)
-`man` page for `fdopen`:
  -Associates a stream with existing file descriptor `fd`
  -Second argument `"a+"`: opens file for reading and appending (writing at end of file)
  -Side note: we can open a normal disk file with `"a+"` as well
-Looking at `handle_connection`:
  -Note that we supply the same `FILE` for the parameters `fin` and `fout`
  -Unlike file descriptors for disk files, socket file descriptors are like 2 streams: the file descriptor is not seekable, and input and output are unrelated
  -For example, consider the pseudo-code:

```
f = open(disk_file, "a+");
fputs("hello", f);
fgets(s, sizeof(s), f);
```

  With a disk file, you would expect to read what you put in; but that's *not* the case with a socket
-Weakness: there is only one *thread of control* in this program
-if the program stalls, how can we find out where it's stalling?
  -use `gdb`: we find out that it's stalled in `fgets` inside `handle_connection`
  -basically, `fgets` won't return until the next line has been read
  -this would be good but sometimes there are multiple "stakeholders", each with their own "goals" (i.e. they are not necessarily on the same page, and cannot be expected to act in ways that are optimal for the server's overall operation)
-How can the stall be fixed?
  -Multithreading? No, we don't know what that is.
  -Fork!
-Using `fork` (see `serviceserver-03.c`):
  -Each process now handles a different connection
  -Need to make sure to `exit` after a connection has been handled - otherwise there'll be a lot of processes hanging around using up resources and not working
-Digression on `execvp`:
  -We can replace the `handle_connection` with a normal shell prompt, running `execvp(argv[0], argv)` on `{"echo", "foo", NULL}`.

    -This works fine. What if `execvp` fails, though - what if we run “`ecko foo`” instead?
    -This is the reason we need to `exit(1);` after `execvp`

-Now, when we change back to the normal `handle_connection`, the fork lets us run `serviceserver` with multiple clients; however, a client should exit once it reads EOF, but it isn’t exiting - what gives?
-Debugging techniques:
    -`printf`: We place a `fprintf` right before the `exit` call. The statement gets printed but it doesn’t seem like the child is exiting
    -`ps`: We can use `ps` to check if the child is exiting (e.g. `ps auxww | grep serviceserver`)
    -`valgrind`: No

-Problem turns out to be that `cfd` (the connection file descriptor) is still open in the parent, so we need to add the line `close(cfd);`
-Checkout `serviceblaster`, which sets up a bunch of connections with `serviceserver`
-Now we have a new issue: a client can no longer DoS attack our server, but we’ve created the problem of the user being able to control our resources. Each time a request is made, the server forks, and new processes are expensive. We can fix this with threading.

Threading:
-A thread is a virtual processor (vs. a process, which is a virtual computer)
-Threads share primary memory, but have different stacks, register sets, and instruction pointers
-How do we create a new thread? We will use the `pthreads` (POSIX threads) library.
-In particular, the function `pthread_create` creates a new thread that begins by running a new function
-Declared as `pthread_create(id, attr, function, arg to function)` -- according to Prof. Kohler the parameter `attr` is almost always `NULL`

Looking at `serviceserver-05.c`:
-Why does a thread start out calling a new function, with a fresh stack, instead of copying memory and jumping into a function in medias res (like `fork` does)? Because, in addition to being more resource efficient, this way allows threads to share the same virtual memory space, which means that they can communicate with each other by accessing shared memory locations (not copies of the same memory). For example, threads can refer to a global variable that is used by other threads
-We can print out the approximate location of the various thread stacks by printing out the location of a local variable in each thread.
    -Stacks are arranged contiguously (growing towards lower addresses), with a big distance between main stack and threads, and between the 1st thread and the 2nd
    -A new thread will be given the space of a thread that has exited (if available)

-Running `serviceblaster`, we find that `pthread_create` eventually runs out of memory - there’s only so much space for stacks.
-Forms of protection:

- -Set a cap on the number of connections ("How about 100?")
- -Set a timeout for connections that have been established more than n seconds ago

-Checkout `serviceserver-06.c`:

- -We use a global variable to track and limit the number of connections
- -But this has terrible CPU utilization! (Try running `./serviceserver-06` with `./serviceblaster` and look at `top`)

-Now checkout `serviceserver-08.c`, which uses a condition variable

-There are two uses for a condition variable:

- -Waiting for the condition to become true (`pthread_cond_wait`)
- -Signaling that the condition has become true (`pthread_cond_signal`; note that these type of signals are not like the interrupt signals we have previously talked about)

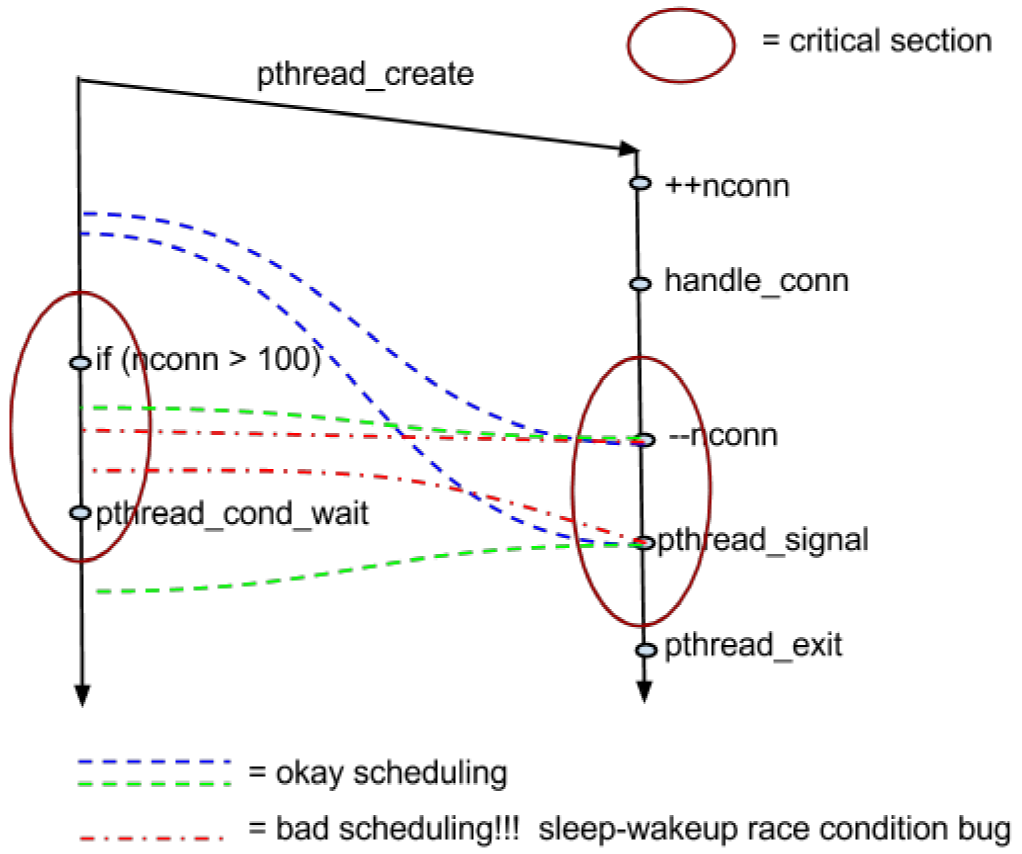
-*Synchronization* is the process of coordinating multiple threads/processes

-A *synchronization object* is an object that helps with synchronization

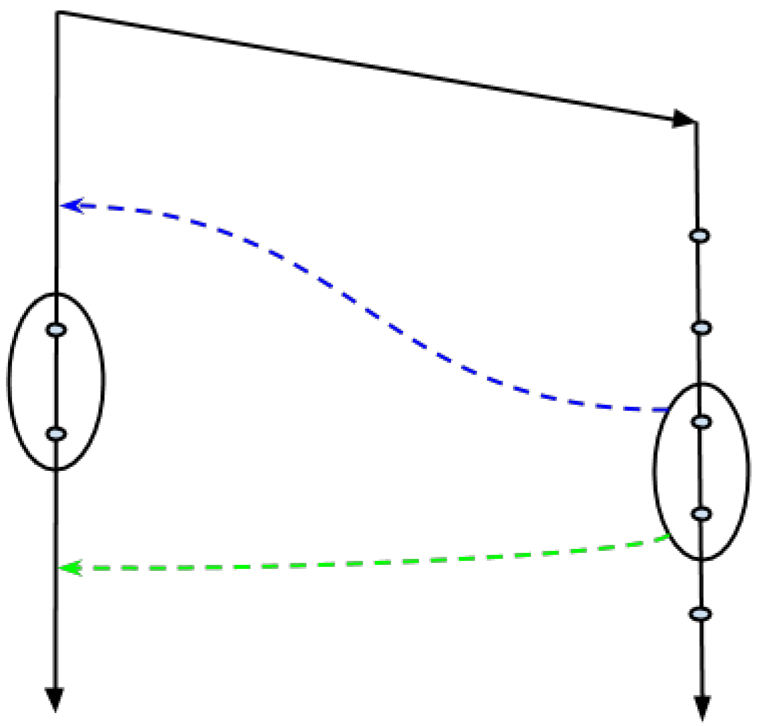
-*Mutual exclusion*: at most one thread allowed in a code region at a time

-Put `pthread_mutex_lock` and `pthread_mutex_unlock` at the beginning and end of critical sections (e.g. mutual exclusion regions)

Here's a synchronization diagram for our threaded `serviceserver` that **does not** implement mutual exclusion regions (`main` is on the left, `connection_thread` on the right):



And here’s a (simplified) synchronization diagram **after** implementing mutual exclusion regions (shown as the large ovals):



Either schedule is fine! The race condition bug has been removed

Here’s the relevant code for those who are interested. First, from `connection_thread`:

```
pthread_mutex_lock(&mutex)
--n_connection_threads;
pthread_cond_signal(&condvar);
pthread_mutex_unlock(&mutex);
pthread_exit(0);
```

And now from `main`:

```
pthread_mutex_lock(&mutex);
if (n_connections_threads > 100)
     pthread_cond_wait(&condvar, &mutex);
pthread_mutex_unlock(&mutex);
```