Lecture 10: From C threads to C++ threads

Principles of Computer Systems

Spring 2019

Stanford University

Computer Science Department

Lecturer: Chris Gregg

PDF of this presentation

Midterm Details

- Midterm on Thursday, May 2, 6pm-8pm, Hewlett 200 (across the hall from the normal lecture hall)
- I will contact students who have conflicts this evening with alternate times / locations
- I will also contact students with accommodations this evening
- The exam will be administered using BlueBook, a computerized testing software that you will run on your laptop. If you don't have a laptop to run the program on, let Chris know ASAP.
  - You can download the BlueBook software from the main CS 110 website.
    - Make sure you test the program out before you come to the exam. We will post a basic test exam later this evening.
  - We will have limited power outlets for laptops, so please ensure you have a charged battery
- You are allowed one back/front page of 8.5 x 11in paper for any notes you would like to bring in. We will also provide a limited reference sheet with functions you may need to use for the exam.

Assignment 4: Stanford Shell

- Assignment 4 is a comprehensive test of your abilities to **`fork`** / **`execvp`** child processes and manage them through the use of signal handlers. It also tests your ability to use pipes.
- You will be writing a shell (demo: **`assign3/samples/stsh_soln`**)
  - The shell will keep a list of all background processes, and it will have some standard shell abilities:
    - you can quit the shell (using **`quit`** or **`exit`**)
    - you can bring them to the front (using **`fg`**)
    - you can continue a background job (using **`bg`**)
    - you can kill a set of processes in a pipeline (using **`slay`**) (this will entail learning about process groups)
    - you can stop a process (using **`halt`**)
    - you can continue a process (using **`cont`**)
    - you can get a list of jobs (using **`jobs`**)
  - You are responsible for creating pipelines that enable you to send output between programs, e.g.,
    - **`ls | grep stsh | cut -d- -f2`**
    - **`sort < stsh.cc | wc > stsh-wc.txt`**
  - You will also be handing off terminal control to foreground processes, which is new

Assignment 4: Stanford Shell

- Assignment 4 contains a lot of moving parts!
- Read through all the header files!
- You will only need to modify **`stsh.cc`**
- You can test your shell programmatically with **`samples/stsh-driver`**
- One of the more difficult parts of the assignment is making sure you are keeping track of all the processes you've launched correctly. This involves careful use of a **`SIGCHLD`** handler.
  - You will also need to use a handler to capture **`SIGTSTP`** and **`SIGINT`** to capture ctrl-Z and ctrl-C, respectively (notice that these don't affect your regular shell -- they shouldn't affect your shell, either).
- Another tricky part of the assignment is with the piping between processes. It takes time to understand what we are requiring you to accomplish
- There is a very good list of milestones in the assignment -- try to accomplish regular milestones, and you should stay on track.
- I understand that this is a detailed assignment, with a midterm in the middle. I suggest at least starting the assignment before the midterm and getting through a couple of milestones. But, also take the time to study for the midterm.

Lecture 10: From C threads to C++ threads

- Introverts Revisited, in C++
  - Rather than deal with **pthreads** as a platform-specific extension of C, I'd rather use a thread package that's officially integrated into the language itself.
    - As of 2011, C++ provides support for threading and many synchronization directives.
    - Because C++ provides better alternatives for generic programming than C does, we avoid the **void** * tomfoolery required when using **pthreads** .
  - Presented below is the object-oriented C++ equivalent of the **introverts** example we've already seen once before. The full program is online right here.

```
static void recharge() {
    cout << oslock << "I recharge by spending time alone." << endl << osunlock;
}

static const size_t kNumIntroverts = 6;
int main(int argc, char *argv[]) {
  cout << "Let's hear from " << kNumIntroverts << " introverts." << endl
  thread introverts[kNumIntroverts]; // declare array of empty thread handles
  for (thread& introvert: introverts)
     introvert = thread(recharge);    // move anonymous threads into empty handles
  for (thread& introvert: introverts)
     introvert.join();
  cout << "Everyone's recharged!" << endl;
  return 0;
}
```

Lecture 10: From C threads to C++ threads

- We declare an array of empty **thread** handles as we did in the equivalent C version.
- We install the **recharge** function into temporary threads that are then moved (via the **thread**'s **operator=(thread&& other)**) into a previously empty **thread** handle.
  - This is a relatively new form of **operator=** that fully transplants the contents of the **thread** on the right into the **thread** on the left, leaving the **thread** on the right fully gutted, as if it were zero-arg constructed. Restated, the left and right thread objects are effectively swapped.
  - This is an important distinction, because a traditional **operator=** would produce a second working copy of the same **thread**, and we don't want that.
- The **join** method is equivalent to the **pthread_join** function we've already discussed.
- The prototype of the thread routine—in this case, **recharge**—can be anything (although the return type is always ignored, so it should generally be **void**).
- **operator<<**, unlike **printf**, isn't thread-safe.
  - Jerry Cain has constructed custom stream manipulators called **oslock** and **osunlock** that can be used to acquire and release exclusive access to an **ostream**.
  - These manipulators—which we can use by **#include**-ing **"ostreamlock.h"**—can be used to ensure at most one thread has permission to write into a stream at any one time.

Lecture 10: From C threads to C++ threads

- Thread routines can accept any number of arguments using variable argument lists. (Variable argument lists—the C++ equivalent of the ellipsis in C—are supported via a recently added feature called variadic templates.)
- Here's a slightly more involved example, where **greet** threads are configured to say hello a variable number of times.

```
static void greet(size_t id) {
  for (size_t i = 0; i < id; i++) {
    cout << oslock << "Greeter #" << id << " says 'Hello!'" << endl << osunlock;
    struct timespec ts = {
      0, random() % 1000000000
    };
    nanosleep(&ts, NULL);
  }
  cout << oslock << "Greeter #" << id << " has issued all of his hellos, "
       << "so he goes home!" << endl << osunlock;
}

static const size_t kNumGreeters = 6;
int main(int argc, char *argv[]) {
  cout << "Welcome to Greetland!" << endl;
  thread greeters[kNumGreeters];
  for (size_t i = 0; i < kNumGreeters; i++) greeters[i] = thread(greet, i + 1);
  for (thread& greeter: greeters) greeter.join();
  cout << "Everyone's all greeted out!" << endl;
  return 0;
}
```

Lecture 10: From C threads to C++ threads

- Multiple threads are often spawned to subdivide and collectively solve a larger problem. Full program is right here.
- Consider the scenario where 10 ticket agents answer telephones—as they might have before the internet came along—at United Airlines to jointly sell 250 airline tickets.
  - Each ticket agent answers the telephone, and each telephone call always leads to the sale of precisely one ticket.
  - Rather than requiring each ticket agent sell 10% of the tickets, we'll account for the possibility that some ticket sales are more time consuming than others, some ticket agents need more time in between calls, etc. Instead, we'll require that all ticket agents keep answering calls and selling tickets until all have been sold.
- Here's our first stab at a **main** function.

```
int main(int argc, const char *argv[]) {
  thread agents[10];
  size_t remainingTickets = 250;
  for (size_t i = 0; i < 10; i++)
    agents[i] = thread(ticketAgent, 101 + i, ref(remainingTickets));
  for (thread& agent: agents) agent.join();
  cout << "End of Business Day!" << endl;
  return 0;
}
```

Lecture 10: From C threads to C++ threads

- As with most multithreaded programs, the main thread elects to spawn child threads to subdivide and collaboratively solve the full problem at hand.
  - In this case, the **main** function declares the master copy of the remaining ticket count—aptly named **remainingTickets**—and initializes it to 250.
  - The main thread then spawns ten child threads to run some **ticketAgent** thread routine, yet to be fully defined. Each agent is assigned a unique id number between 101 and 110, inclusive, and a reference to **remainingTickets** is shared with each thread.
  - As is typical, the main thread blocks until all child threads have finished before exiting. Otherwise, the entire process might be torn down even though some child threads haven't finished.

```
int main(int argc, const char *argv[]) {
  thread agents[10];
  size_t remainingTickets = 250;
  for (size_t i = 0; i < 10; i++)
    agents[i] = thread(ticketAgent, 101 + i, ref(remainingTickets));
  for (thread& agent: agents) agent.join();
  cout << "End of Business Day!" << endl;
  return 0;
}
```

Lecture 10: From C threads to C++ threads

- The **ticketAgent** thread routine accepts an id number (used for logging purposes) and a reference to the **remainingTickets**.
- It continually polls **remainingTickets** to see if any tickets remain, and if so, proceeds to answer the phone, sell a ticket, and publish a little note about the ticket sale to **cout**.
- **handleCall**, **shouldTakeBreak**, and **takeBreak** are all in place to introduce short, random delays and guarantee that each test run is different than prior ones. Full program is right here.

```
static void ticketAgent(size_t id, size_t& remainingTickets) {
  while (remainingTickets > 0) {
    handleCall(); // sleep for a small amount of time to emulate conversation time.
    remainingTickets--;
    cout << oslock << "Agent #" << id << " sold a ticket! (" << remainingTickets
     << " more to be sold)." << endl << osunlock;
    if (shouldTakeBreak()) // flip a biased coin
      takeBreak();         // if comes up heads, sleep for a random time to take a break
  }
  cout << oslock << "Agent #" << id << " notices all tickets are sold, and goes home!"
       << endl << osunlock;
}
```

Lecture 10: From C threads to C++ threads

- Presented below right is the abbreviated output of a **confused-ticket-agents** run.
- In its current state, the program suffers from a serious race condition.
- Why? Because **remainingTickets > 0** test and **remainingTickets--** aren't guaranteed to execute within the same time slice.
- If a thread evaluates **remainingTickets > 0** to be **true** and commits to selling a ticket, the ticket might not be there by the time it executes the decrement. That's because the thread may be swapped off the CPU after the decision to sell but before the sale, and during the dead time, other threads—perhaps the nine others—all might get the CPU and do precisely the same thing.
- The solution? Ensure the decision to sell and the sale itself are executed without competition.

```
cgregg@myth55$ ./confused-ticket-agents
Agent #110 sold a ticket! (249 more to be sold).
Agent #104 sold a ticket! (248 more to be sold).
Agent #106 sold a ticket! (247 more to be sold).
// some 245 lines omitted for brevity
Agent #107 sold a ticket! (1 more to be sold).
Agent #103 sold a ticket! (0 more to be sold).
Agent #105 notices all tickets are sold, and goes home!
Agent #104 notices all tickets are sold, and goes home!
Agent #108 sold a ticket! (4294967295 more to be sold).
Agent #106 sold a ticket! (4294967294 more to be sold).
Agent #102 sold a ticket! (4294967293 more to be sold).
Agent #101 sold a ticket! (4294967292 more to be sold).
// carries on for a very, very, very long time
```

Lecture 10: From C threads to C++ threads

- Before we solve this problem, we should really understand why **remainingTickets--** itself isn't even thread-safe.
  - C++ statements aren't inherently atomic. Virtually all C++ statements—even ones as simple as **remainingTickets--**—compile to multiple assembly code instructions.
  - Assembly code instructions are atomic, but C++ statements are not.
  - **g++** on the myths compiles **remainingTickets--** to five assembly code instructions, as with:

```
0x0000000000401a9b <+36>:    mov    -0x20(%rbp),%rax
0x0000000000401a9f <+40>:    mov    (%rax),%eax
0x0000000000401aa1 <+42>:    lea    -0x1(%rax),%edx
0x0000000000401aa4 <+45>:    mov    -0x20(%rbp),%rax
0x0000000000401aa8 <+49>:    mov    %edx,(%rax)
```

  - The first two lines drill through the **ticketsRemaining** reference to load a copy of the **ticketsRemaining** held in **main**. The third line decrements that copy, and the last two write the decremented copy back to the **ticketsRemaining** variable held in **main**.

Lecture 10: From C threads to C++ threads

- Before we solve this problem, we should really understand why **remainingTickets--** itself isn't even thread-safe.
  - If a thread executes the first three (or two, or four) of these instructions and gets swapped off the CPU, it will eventually get the processor back, but by that time the local copy will be outdated if one or more other threads have since gotten processor time and executed some or all of the same five assembly code instructions below.
  - The problem: ALU operations are executed on copies of shared data

```
0x0000000000401a9b <+36>:    mov    -0x20(%rbp),%rax
0x0000000000401a9f <+40>:    mov    (%rax),%eax
0x0000000000401aa1 <+42>:    lea    -0x1(%rax),%edx
0x0000000000401aa4 <+45>:    mov    -0x20(%rbp),%rax
0x0000000000401aa8 <+49>:    mov    %edx,(%rax)
```

  - A specific pathological example: **main**'s copy of **remainingTickets** is 250, and a thread manages to execute the first three of these five instructions before before swapped off the CPU. **%edx** stores a 249, but the thread has yet to push that 249 back to **main**'s **remainingTickets**. It will when it gets the processor again, but it might not get the processor until the other threads have cooperatively sold one more tickets (or perhaps even all of them).

Lecture 10: From C threads to C++ threads

- We need to guarantee that the code that tests for remaining tickets, sells a ticket, and everything in between are executed as part of one large transaction, without interference from other threads. Restated, we must guarantee that at no other threads are permitted to even **examine** the value of **ticketsRemaining** if another thread is staged to modify it.
- One solution: provide a directive that allows a thread to ask that it not be swapped off the CPU while it's within a block of code that should be executed transactionally.
  - That, however, is not an option, and shouldn't be.
  - That would grant too much power to threads, which could abuse the option and block other threads from running for an indeterminate amount of time.
- The other option is to rely on a concurrency directive that can be used to prevent more than one thread from being anywhere in the same critical region at one time. That concurrency directive is the **mutex**, and in C++ it looks like this:

```
class mutex {
public:
  mutex();         // constructs the mutex to be in an unlocked state
  void lock();     // acquires the lock on the mutex, blocking until it's unlocked
  void unlock();   // releases the lock and wakes up another threads trying to lock it
};
```

Lecture 10: From C threads to C++ threads

- The name **mutex** is just a contraction of the words *mutual* and *exclusion*. It's so named because its primary use it to mark the boundaries of a critical region—that is, a stretch of code where at most one thread is permitted to be at any one moment.
  - Restated, a thread executing code within a critical region enjoys exclusive access.
- The constructor initializes the **mutex** to be in an unlocked state.
- The **lock** method will *eventually* acquire a lock on the **mutex**.
  - If the **mutex** is in an unlocked state, **lock** will lock it and return immediately.
  - If the **mutex** is in a locked state (presumably because another thread called **lock** but has yet to **unlock**), **lock** will pull the calling thread off the CPU and render it ineligible for processor time until notified the lock on the **mutex** was released.
- The **unlock** method will release the lock on a mutex. The only thread qualified to release the lock on the **mutex** is the one that holds the lock.

```
class mutex {
public:
  mutex();         // constructs the mutex to be in an unlocked state
  void lock();     // acquires the lock on the mutex, blocking until it's unlocked
  void unlock();   // releases the lock and wakes up another threads trying to lock it
};
```

Lecture 10: From C threads to C++ threads

- We can declare a single **mutex** aside the declaration of **remainingTickets** in the **main** function, and we can use that **mutex** to mark the boundaries of the critical region.
- This requires the **mutex** also be shared by reference with the **ticketAgent** thread routine so that all child threads compete to acquire the same lock.
- The new **ticketAgent** thread routine looks like this:

```
static void ticketAgent(size_t id, size_t& remainingTickets, mutex& ticketsLock) {
  while (true) {
    ticketsLock.lock();
    if (remainingTickets == 0) break;
    handleCall();
    remainingTickets--;
    cout << oslock << "Agent #" << id << " sold a ticket! (" << remainingTickets
         << " more to be sold)." << endl << osunlock;
    ticketsLock.unlock();
    if (shouldTakeBreak()) takeBreak();
  }
  ticketsLock.unlock();
  cout << oslock << "Agent #" << id << " notices all tickets are sold, and goes home!"
       << endl << osunlock;
}
```

Lecture 10: From C threads to C++ threads

- The **main** function might look like the one I present below (and right here).
  - **main** declares a single **mutex** called **ticketLock**. I can call it anything I want to, but I want to be clear that it's in place to guard any potential surgery on a variable with **tickets** in its name. That's why I choose the name I do.
  - Because the **ticketAgent** accepts a reference to a **mutex**, the **thread** constructor needs to pass a reference to a **mutex** via an addition argument that wasn't present in the first version. That's what **ref(ticketLock)** is all about.
  - Fundamentally, the **main** function maintains the state information needed to enable all child threads to collaboratively work in parallel without introducing any race conditions.

```
int main(int argc, const char *argv[]) {
  size_t remainingTickets = 250;
  mutex ticketsLock;
  thread agents[10];
  for (size_t i = 0; i < 10; i++)
    agents[i] = thread(ticketAgent, 101 + i, ref(remainingTickets), ref(ticketsLock));
  for (thread& agent: agents) agent.join();
  cout << "End of Business Day!" << endl;
  return 0;
}
```

Lecture 10: From C threads to C++ threads

- There is still one issue with this strategy -- there is an inherent "waiting around" for agents who want to sell tickets but can't because of the lock.
- The way we've set it up, only one ticket agent can sell at a time!
  - Yes, there is some parallelism with the break-taking, but the ticket-selling is serialized.
- Can we do better?
  - Well, it depends -- as it stands, the only way a ticket can get marked as sold is after it has indeed been sold (line 6 below).
  - If we assume that a ticket is *always* sold, then we can do much better.

```
1  static void ticketAgent(size_t id, size_t& remainingTickets, mutex& ticketsLock) {
2    while (true) {
3      ticketsLock.lock();
4      if (remainingTickets == 0) break;
5      handleCall();
6      remainingTickets--;
7      cout << oslock << "Agent #" << id << " sold a ticket! (" << remainingTickets
8           << " more to be sold)." << endl << osunlock;
9      ticketsLock.unlock();
10     if (shouldTakeBreak()) takeBreak();
11   }
12   ticketsLock.unlock();
13   cout << oslock << "Agent #" << id << " notices all tickets are sold, and goes home!"
14        << endl << osunlock;
15 }
```

Lecture 10: From C threads to C++ threads

- Here is our updated ticketAgent function:

```
 1 static void ticketAgent(size_t id, size_t& remainingTickets, mutex& ticketsLock) {
 2   while (true) {
 3     ticketsLock.lock();
 4     if (remainingTickets == 0) break;
 5     remainingTickets--;
 6     ticketsLock.unlock();
 7     handleCall(); // assume this sale is successful
 8     cout << oslock << "Agent #" << id << " sold a ticket! (" << remainingTickets
 9          << " more to be sold)." << endl << osunlock;
10     if (shouldTakeBreak()) takeBreak();
11   }
12   ticketsLock.unlock();
13   cout << oslock << "Agent #" << id << " notices all tickets are sold, and goes home!"
14        << endl << osunlock;
15 }
```

- Now, we allow each agent to grab an available ticket, but they must sell it!
- If an agent did not sell the ticket, some other agents might go home early.
  - But, that might be okay for our model.
  - If not, we would have to think about how to ensure that the agent does remain until their ticket is sold, meaning we would have to have a bit more logic.