Lecture 10: Threads and Mutexes

Principles of Computer Systems

Autumn 2019

Stanford University

Computer Science Department

Lecturers: Chris Gregg and

Philip Levis

PDF of this presentation

Midterm Details

- Midterm on Monday, October 28, in class
- We will also contact students with accommodations in the next few days
- 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 us know ASAP and we will provide one.
  - 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 in a few days.
  - 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.
  - Knowing the exact order of the arguments to system calls we've covered isn't expected, but knowing their semantics is

pthreads in C (review of last lecture)

- In C, threads are a library, called pthreads, which comes with all standard UNIX installations of **gcc**
  - The primary **pthreads** data type is the **pthread_t**, which is an opaque type used to manage the execution of a function within its own thread of execution.
  - In the previous lecture, you saw two functions, **pthread_create** and **pthread_join**.

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

int pthread_join(pthread_t thread, void **retval);
```

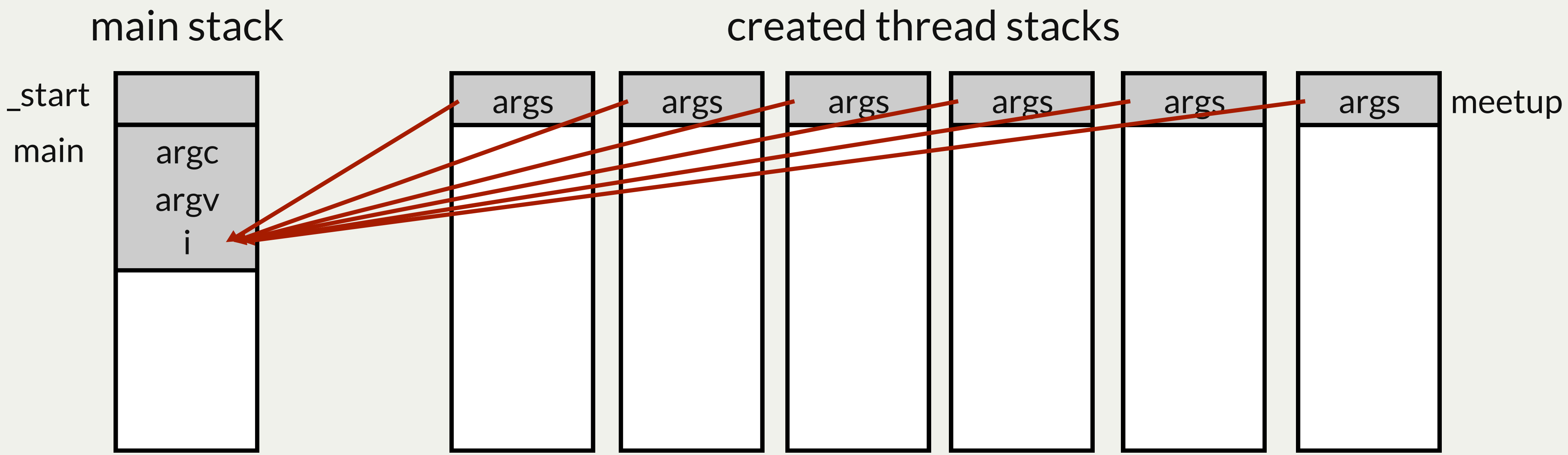
- With pthreads, you pass a function with signature **void* f(void* input)**, the library allocates a stack and runs the thread
- Threads all share the address space of a single process: you need to be very careful about how they share data, similarly to how we did for signal handlers

```c
1     for (size_t i = 0; i < kNumFriends; i++)
2         pthread_create(&friends[i], NULL, meetup, &i);
3     for (size_t j = 0; j < kNumFriends; j++)
4         pthread_join(friends[j], NULL);
```

bug on line 2!

pthread Bug in Last Lecture

```
for (size_t i = 0; i < kNumFriends; i++)
    pthread_create(&friends[i], NULL, meetup, &i);
for (size_t j = 0; j < kNumFriends; j++)
    pthread_join(friends[j], NULL);
```



- Solve problem by passing each thread a pointer to its associated string, which doesn't change

pthreads are great, but...

- Because they are a library, with pthreads you have to do everything manually (much like signals)
- For example, a common way to implement a critical section is through a *mutual exclusion* variable (mutex)
  - A mutex is a lock: take the lock before entering the critical section and release it after
  - If a thread tries to take a locked lock, it waits until it is unlocked (like how we blocked signals)
  - If you forget to unlock the lock, everyone else waits forever (deadlock!)
- C++'s greater guarantees on when things occur allow us to avoid some common errors
  - We'll start by showing you the basic APIs, so you can see how things can go wrong, then show other supported approaches that help

```
uint64_t increment_counter(void) {
  pthread_mutex_lock(&counter_lock);
  counter++;
  uint64_t val = counter;
  pthread_mutex_unlock(&counter_lock);
  return val;
}
```

pthread approach

If you forget to unlock, deadlock

```
uint64_t increment_counter(void) {
  lock_guard<mutex> lg(&counter_lock);
  counter++;
  uint64_t val = counter;
  return val;
}
```

C++ approach

Can't forget to unlock!

C++ Threads

```
static void *recharge(void *args) {
    printf("I recharge by spending time alone.\n");
    return NULL;
}

static const size_t kNumIntroverts = 6;
int main(int argc, char *argv[]) {
    printf("Let's hear from %zu introverts.\n", kNumIntroverts);
    pthread_t introverts[kNumIntroverts];
    for (size_t i = 0; i < kNumIntroverts; i++)
        pthread_create(&introverts[i], NULL, recharge, NULL);
    for (size_t i = 0; i < kNumIntroverts; i++)
        pthread_join(introverts[i], NULL);
    printf("Everyone's recharged!\n");
    return 0;
}
```

C/pthreads

```
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;
}
```

C++

Details on the Code: It’s Subtle

```
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();
```

- We create a thread that executes the **recharge** function and return a thread handle to it
- We then move the thread handle (via the **thread**'s **operator=(thread&& other)**) into the array
  - This is a different meaning for **operator=**
  - After it executes, the right hand side is an empty thread
    - thread t1 = thread(func);
    - thread t2 = t1; // t1 is no longer a handle for the thread created
  - This is an important distinction, because a traditional **operator=** would produce a second working copy of the same **thread**, which would be bad in so many ways (share a stack???)
- 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**).

WARNING: Thread Safety and Standard I/O

- **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.

No More Void* Tomfoolery

- 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;
}
```

Thread-Level Parallelism

- Threads allow a process to parallelize a problem across multiple cores
- Consider a scenario where we want to process 250 images and have 10 cores
- Completion time is determined by the slowest thread, so we want them to have equal work
  - Static partitioning: just give each thread 25 of the images to process. Problem: what if some images take much longer than others?
  - Work queue: have each thread fetch the next unprocessed image
- Here's our first stab at a **main** function.

```
int main(int argc, const char *argv[]) {
  thread processors[10];
  size_t remainingImages = 250;
  for (size_t i = 0; i < 10; i++)
    processors[i] = thread(process, 101 + i, ref(remainingImages));
  for (thread& proc: processors) proc.join();
  cout << "Images done!" << endl;
  return 0;
}
```

Thread Function

- The **processor** thread routine accepts an id number (used for logging purposes) and a reference to the **remainingImages**.
- It continually checks **remainingImages** to see if any images remain, and if so, processes the image and sends a message to **cout**
- **processImage** execution time depends on the image.
- Note how we can declare a function that takes a **size_t** and a **size_t&** as arguments

```
static void process(size_t id, size_t& remainingImages) {
  while (remainingImages > 0) {
    processImage(remainingImages);
    remainingImages--;
    cout << oslock << "Thread#" << id << " processed an image (" << remainingImages
     << " remain)." << endl << osunlock;
  }
  cout << oslock << "Thread#" << id << " sees no remaining images and exits."
       << endl << osunlock;
}
```

- Discuss with your neighbor -- what's wrong with this code?

Race Condition

- Presented below right is the abbreviated output of a **imagethreads** run.
- In its current state, the program suffers from a serious race condition.
- Why? Because **remainingImages > 0** test and **remainingImages--** aren't atomic
- If a thread evaluates **remainingImages > 0** to be **true** and commits to processing an image, the image may have been claimed by another thread.
- This is a concurrency problem!
- Solution? Make the test and decrement *atomic* with a *critical section*
- Atomicity: externally, the code has either executed or not; external observers do not see any intermediate states mid-execution

```
myth60 ~../cs110/cthreads -> ./imagethreads
Thread# 109 processed an image, 249 remain
Thread# 102 processed an image, 248 remain
Thread# 101 processed an image, 247 remain
Thread# 104 processed an image, 246 remain
Thread# 108 processed an image, 245 remain
Thread# 106 processed an image, 244 remain
// 241 lines removed for brevity
Thread# 110 processed an image, 3 remain
Thread# 103 processed an image, 2 remain
Thread# 105 processed an image, 1 remain
Thread# 108 processed an image, 0 remain
Thread# 105 processed an image, 18446744073709551615 remain
Thread# 109 processed an image, 18446744073709551614 remain
```

Why Test and Decrement Is REALLY NOT Thread-Safe

- C++ statements aren't inherently atomic. Virtually all C++ statements—even ones as simple as **remainingImages--**—compile to multiple assembly code instructions.
- Assembly code instructions are atomic, but C++ statements are not.
- **g++** on the myths compiles **remainingImages--** 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 **remainingImages** reference to load a copy of the **remainingImages** held on **main**'s stack. The third line decrements that copy, and the last two write the decremented copy back to the **remainingImages** variable held on **main**'s stack.
- The ALU operates on registers, but registers are private to a core, so the variable needs to be loaded from and stored to memory.
  - Each thread makes a local copy of the variable before operating on it
  - What if multiple threads all load the variable at the same time: they all think there's only 128 images remaining and process 128 at the same time

Mutual Exclusion

- A mutex is a type used to enforce *mutual exclusion*, i.e., a critical section
- Mutexes are often called locks
  - To be very precise, mutexes are one kind of lock, there are others (read/write locks, reentrant locks, etc.), but we can just call them locks in this course, usually "lock" means "mutex"
- When a thread locks a mutex
  - If the lock is unlocked the thread takes the lock and continues execution
  - If the lock is locked, the thread blocks and waits until the lock is unlocked
  - If multiple threads are waiting for a lock they all wait until lock is unlocked, one receives lock
- When a thread unlocks a mutex
  - It continues normally; one waiting thread (if any) takes the lock and is scheduled to run
- This is a subset of the C++ mutex abstraction: nicely simple!

```
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
};
```

Building a Critical Section with a Mutex

- **main** instantiates a mutex, which it passes (by reference!) to invocations of **process.**
- The **process** code uses this lock to protect **remainingImages**.
- Note we need to unlock on line 5 -- in complex code forgetting this is an easy bug

```
1  static void process(size_t id, size_t& remainingImages, mutex& counterLock) {
2    while (true) {
3      counterLock.lock();
4      if (remainingImages == 0) {
5        counterLock.unlock();
6        break;
7      }
8      processImage(remainingImages);
9      remainingImages--;
10     cout << oslock << "Thread#" << id << " processed an image (" << remainingImages
11      << " remain)." << endl << osunlock;
12     counterLock.unlock();
13   }
14   cout << oslock << "Thread#" << id << " sees no remaining images and exits."
15   << endl << osunlock;
16 }
17
18 int main(int argc, const char *argv[]) {
19   size_t remainingImages = 250;
20   mutex  counterLock;
21   thread processors[10];
22   for (size_t i = 0; i < 10; i++)
23     agents[i] = thread(process, 101 + i, ref(remainingImages), ref(counterLock));
24   for (thread& agent: agents) agent.join();
25   cout << "Done processing images!" << endl;
26   return 0;
27 }
```

Critical Sections Can Be a Bottleneck

- The way we've set it up, only one thread agent can process an image at a time!
  - Image processing is actually serialized
- We can do better: serialize deciding which image to process and parallelize the actual processing
- Keep your critical sections as small as possible!

```cpp
static void process(size_t id, size_t& remainingImages, mutex& counterLock) {
  while (true) {
    size_t myImage;

    counterLock.lock();     // Start of critical section
    if (remainingImages == 0) {
      counterLock.unlock(); // Rather keep it here, easier to check
      break;
    } else {
      myImage = remainingImages;
      remainingImages--;
      counterLock.unlock(); // end of critical section

      processImage(myImage);
      cout << oslock << "Thread#" << id << " processed an image (" << remainingImages
      << " remain)." << endl << osunlock;
    }
  }
  cout << oslock << "Thread#" << id << " sees no remaining images and exits."
  << endl << osunlock;
}
```

Problems That Might Arise

- What if **processImage** can return an error?
  - E.g., what if we need to distinguish allocating an image and processing it
  - A thread can grab the image by decrementing **remainingImages** but if it fails there's no way for another thread to retry
  - Because these are threads, if one thread has a SEGV the whole process will fail
  - A more complex approach might be to maintain an actual queue of images and allow threads (in a critical section) to push things back into the queue
- What if image processing times are *highly* variable (e.g, one image takes 100x as long as the others)?
  - Might scan images to estimate execution time and try more intelligent scheduling
- What if there's a bug in your code, such that sometimes processImage randomly enters an infinite loop?
  - Need a way to reissue an image to an idle thread
  - An infinite loop of course shouldn't occur, but when we get to networks sometimes execution time can vary by 100x for reasons outside our control

Some Types of Mutexes

- Standard **mutex**: what we've seen
  - If a thread holding the lock tries to re-lock it, deadlock
- **recursive_mutex**
  - A thread can lock the mutex multiple times, and needs to unlock it the same number of times to release it to other threads
- **timed_mutex**
  - A thread can **try_lock_for** / **try_lock_until**: if time elapses, don't take lock
  - Deadlocks if same thread tries to lock multiple times, like standard mutex
- In this class, we'll focus on just regular **mutex**

How Do Mutexes Work?

- Something we've seen a few times is that you can't read and write a variable atomically
  - But a mutex does so! If the lock is unlocked, lock it
- How does this work with caches?
  - Each core has its own cache
  - Writes are typically write-back (write to higher cache level when line is evicted), not write-through (always write to main memory) for performance
  - Caches are *coherent* -- if one core writes to a cache line that is also in another core's cache, the other core's cache line is invalidated: this can become a performance problem
- Hardware provides atomic memory operations, such as compare and swap
  - cas old, new, addr
    - If addr == old, set addr to new
  - Use this as a single bit to see if the lock is held and if not, take it
  - If the lock is held already, then enqueue yourself (in a thread safe way) and tell kernel to sleep you
  - When a node unlocks, it clears the bit and wakes up a thread

Questions about threads, mutexes, race conditions, or critical sections?

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.