Lecture 12: More on Multithreading, CVs, and Semaphores

Principles of Computer Systems

Spring 2019

Stanford University

Computer Science Department

Lecturer: Chris Gregg

PDF of this presentation

Midterm: Overall, very good!

- You should have received your midterm grade back
- Overall, students did well, and I think it was a challenging exam
- Problem 1d was meant to be difficult and we were looking for a detailed (yet succinct) answer. Very few people got it correct (but it is worth reviewing to see what really is going on).
- Problem 1c: many people did not **`waitpid`** for the children to finish (and we've received some regrade requests saying that we didn't ask you to do that, which we have denied). This is what would happen without the **`waitpid`**s:

```
$ cat testInput.txt
carrot
apple
banana
cgregg@myth54:~/cs110/spring-2019/midterm/twoOutput$ ./two-output sort wc < testInput.txt
cgregg@myth54:~/cs110/spring-2019/midterm/twoOutput$       3       3      20
apple
banana
carrot
```

- Notice that the prompt returns immediately...that is something you *always* want to avoid. Because you were writing the **`main`** function and knew all other functions that would be called, it should have been understood that the parent needed to wait for the children.

Lecture 12: More on Multithreading, CVs, and Semaphores

- Review from last week
  - We now have *three* distinct ways to coordinate between threads:
    - **`mutex`**
    - **`condition_variable_any`**
    - **`semaphore`** (which is not part of the C++ standard, and easy to write)
  - Only use the most minimal one you need, even though another may work as well.
  - Let's review all three locking structures

Lecture 12: mutex

- A mutex is a simple lock that is shared between threads, used to protect critical regions of code or shared data structures.
- It is declared as follows:
  - `mutex m;`
- There are only two functions associated with a mutex:
  - `mutex.lock()`
  - `mutex.unlock()`
- You must pass a mutex by reference (or by pointer) to each thread.
- When a thread attempts to lock a mutex, there are two possibilities:
  - The thread obtains the lock, and executes the next line of code
  - The thread *does not* obtain the lock, and blocks until the lock is released by the current lock-holder, at which point it attempts to obtain the lock again (and could race with other waiting threads).
- Only the current lock-holder is allowed to unlock a mutex
- Mutexes should be held for as short a time as is necessary to protect the region or data structure.
- Deadlock is not strictly possible with a single mutex, although if a thread does not give up a lock, it can cause a program to stop.

Lecture 12: lock_guard<mutex>

- A very nice helper class that we often use with a **`mutex`** is the **`lock_guard<mutex>`**
- The **`lock_guard<mutex>`** is very simple: it obtains the lock in its constructor, and releases the lock in its destructor.
- We use a **`lock_guard<mutex>`** so we don't have to worry about unlocking a **`mutex`** when we leave a locked section of code as it is done for us.
- Example:

```
void function(mutex &m) {
    lock_guard<mutex> lg(m); // m is a mutex we want to lock
    // now the mutex has been locked
    while (true) {
      if (condition1) return; // lock is automatically unlocked in return,
                              // when lg goes out of scope
      // other code
      if (condition2) break;
    }
    // more code
    // mutex will be unlocked after this line when lg goes out of scope
}
```

- Using a lock guard is a good idea when you know you will exit the function at the time you want to release the lock. If you need to release the lock before the function ends, just using a mutex is a better idea.

Lecture 12: conditional_variable_any

- The **`conditional_variable_any`** is a lock that enables one thread to signal to other threads that they may continue if they are waiting for a condition.
- A **`conditional_variable_any`** works *in conjunction with* a **`mutex`** so you must also have a mutex that is associated with the lock.
- First, the user locks the mutex. Then, the condition is checked, and **`cv.wait(m)`** is called. At this point, *the lock is unlocked after the thread is pushed off the processor.*
- When the thread receives a notification, it re-acquires the lock and wakes from **`wait`**.
- The general pattern is as follows:

```
static void waitForPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
  lock_guard<mutex> lg(m);
  while (permits == 0) cv.wait(m);
  permits--;
}

static void grantPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
  lock_guard<mutex> lg(m);
  permits++;
  if (permits == 1) cv.notify_all();
}
```

- You can release a lock before notifying other processes (it is a slight optimization, but not necessary)
- It is fine to have a waiting thread also notify after it completes its critical section.

Lecture 12: conditional_variable_any

- A second form of **`wait()`** is useful because the while loop is very common:

```
template <Predicate pred>
void condition_variable_any::wait(mutex& m, Pred pred) {
  while (!pred()) wait(m);
}
```

- The predicate is a function that returns true or false. We often use a lambda function for the predicate:

```
static void waitForPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
  lock_guard<mutex> lg(m);
  cv.wait(m, [&permits] { return permits > 0; });
  permits--;
}
```

Lecture 12: semaphore

- The **`semaphore`** class is not built in to C++, but it is a useful way to generalize the "permits" idea. We will link against our version of a semaphore for this class, but you should understand how it is built.
- Using a **`semaphore`** is straightforward: you first declare a semaphore with a number of permits you would like:

```
semaphore permits(5); // this will allow five permits
```

- When a thread wants to use a permit, it first **`wait`**s for the permit, and then **`signal`**s when it is done using a permit:

```
permits.wait(); // if five other threads currently hold permits, this will block

// only five threads can be here at once

permits.signal(); // if other threads are waiting, a permit will be available
```

- A **`mutex`** is kind of like a special case of a semaphore with one permit, but you should use a **`mutex`** in that case as it is simpler and more efficient. Additionally, the benefit of a mutex is that it can *only* be released by the lock-holder.

Lecture 12: semaphore

- Question: what would a **`semaphore`** initialized with 0 mean?

```
semaphore permits(0);
```

Lecture 12: semaphore

- Question: what would a **`semaphore`** initialized with 0 mean?

```
semaphore permits(0);
```

- In this case, we don't have *any* permits!
- So, **`permits.wait()`** *always* has to wait for a signal, and will never stop waiting until that signal is received.
- We will see an example of this shortly.

- What about a *negative initializer* for a semaphore?

```
semaphore permits(-9);
```

Lecture 12: semaphore

- What about a *negative initializer* for a semaphore?

```
semaphore permits(-9);
```

- In this case, the semaphore would have to *reach 1* before the wait would stop waiting. You might want to wait until a bunch of threads finished before a final thread is allowed to continue. Example (full program here):

```
1  void writer(int i, semaphore &s) {
2      cout << oslock << "Sending signal " << i << endl << osunlock;
3      s.signal();
4  }
5
6  void read_after_ten(semaphore &s) {
7      s.wait();
8      cout << oslock << "Got enough signals to continue!" << endl << osunlock;
9  }
10
11 int main(int argc, const char *argv[]) {
12     semaphore negSemaphore(-9);
13     thread readers[10];
14     for (size_t i = 0; i < 10; i++) {
15         readers[i] = thread(writer, i, ref(negSemaphore));
16     }
17     thread r(read_after_ten, ref(negSemaphore));
18     for (thread &t : readers) t.join();
19     r.join();
20     return 0;
21 }
```

Lecture 12: More on Multithreading, CVs, and Semaphores

- New concurrency pattern!
  - **semaphore::wait** and **semaphore::signal** can be leveraged to support a different form of communication: **thread rendezvous**.
  - Thread rendezvous is a generalization of **thread::join**. It allows one thread to stall —via **semaphore::wait**—until another thread calls **semaphore::signal**, often because the signaling thread just prepared some data that the waiting thread needs before it can continue.
- To illustrate when thread rendezvous is useful, we'll implement a simple program without it, and see how thread rendezvous can be used to repair some of its problems.
  - The program has two meaningful threads of execution: one thread publishes content to a shared buffer, and a second reads that content as it becomes available.
  - The program is a nod to the communication in place between a web server and a browser. The server publishes content over a dedicated communication channel, and the browser consumes that content.
  - The program also reminds me of how two independent processes behave when one writes to a pipe, a second reads from it, and how the write and read processes behave when the pipe is full (in principle, a possibility) or empty.

Lecture 12: More on Multithreading, CVs, and Semaphores

- Consider the following program, where concurrency directives have been intentionally omitted. (The full program is right here.)

```
static void writer(char buffer[]) {
  cout << oslock << "Writer: ready to write." << endl << osunlock;
  for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length 8
    char ch = prepareData();
    buffer[i % 8] = ch;
    cout << oslock << "Writer: published data packet with character '"
         << ch << "'." << endl << osunlock;
  }
}

static void reader(char buffer[]) {
  cout << oslock << "\t\tReader: ready to read." << endl << osunlock;
  for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length 8
    char ch = buffer[i % 8];
    processData(ch);
    cout << oslock << "\t\tReader: consumed data packet " << "with character '"
         << ch << "'." << endl << osunlock;
  }
}

int main(int argc, const char *argv[]) {
  char buffer[8];
  thread w(writer, buffer);
  thread r(reader, buffer);
  w.join();
  r.join();
  return 0;
}
```

Lecture 12: More on Multithreading, CVs, and Semaphores

- Here's what works:
  - Because the **main** thread declares a circular buffer and shares it with both children, the children each agree where content is stored.
  - Think of the buffer as the state maintained by the implementation of **pipe**, or the state maintained by an internet connection between a server and a client.
  - The **writer** thread publishes content to the circular buffer, and the **reader** thread consumes that same content as it's written. Each thread cycles through the buffer the same number of times, and they both agree that **i % 8** identifies the next slot of interest.
- Here's what's broken:
  - Each thread runs more or less independently of the other, without consulting the other to see how much progress it's made.
  - In particular, there's nothing in place to inform the **reader** that the slot it wants to read from has meaningful data in it. It's possible the writer just hasn't gotten that far yet.
  - Similarly, there's nothing preventing the **writer** from advancing so far ahead that it begins to overwrite content that has yet to be consumed by the **reader**.

Lecture 12: More on Multithreading, CVs, and Semaphores

- One solution? Maintain two **semaphore**s.
  - One can track the number of slots that can be written to without clobbering yet-to-be-consumed data. We'll call it **emptyBuffers**, and we'll initialize it to 8.
  - A second can track the number of slots that contain yet-to-be-consumed data that can be safely read. We'll call it **fullBuffers**, and we'll initialize it to 0.
- Here's the new **main** program that declares, initializes, and shares the two **semaphore**s.

```
int main(int argc, const char *argv[]) {
  char buffer[8];
  semaphore fullBuffers, emptyBuffers(8);
  thread w(writer, buffer, ref(fullBuffers), ref(emptyBuffers));
  thread r(reader, buffer, ref(fullBuffers), ref(emptyBuffers));
  w.join();
  r.join();
  return 0;
}
```

- The **writer** thread waits until at least one buffer is empty before writing. Once it writes, it'll increment the full buffer count by one.
- The **reader** thread waits until at least one buffer is full before reading. Once it reads, it increments the empty buffer count by one.

Lecture 12: More on Multithreading, CVs, and Semaphores

- Here are the two new thread routines:

```
static void writer(char buffer[], semaphore& full, semaphore& empty) {
  cout << oslock << "Writer: ready to write." << endl << osunlock;
  for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length 8
    char ch = prepareData();
    empty.wait();   // don't try to write to a slot unless you know it's empty
    buffer[i % 8] = ch;
    full.signal();  // signal reader there's more stuff to read
    cout << oslock << "Writer: published data packet with character '"
         << ch << "'." << endl << osunlock;
  }
}

static void reader(char buffer[], semaphore& full, semaphore& empty) {
  cout << oslock << "\t\tReader: ready to read." << endl << osunlock;
  for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length 8
    full.wait();    // don't try to read from a slot unless you know it's full
    char ch = buffer[i % 8];
    empty.signal(); // signal writer there's a slot that can receive data
    processData(ch);
    cout << oslock << "\t\tReader: consumed data packet " << "with character '"
         << ch << "'." << endl << osunlock;
  }
}
```

- The reader and writer rely on these **semaphore**s to inform the other how much work they can do before being necessarily forced off the CPU.
- Thought question: can we rely on just one **semaphore** instead of two? Why or why not?

Lecture 12: Multithreading and Networking

- Implementing **myth-buster**!
  - The **myth-buster** is a command line utility that polls all 16 **myth** machines to determine which is the least loaded.
    - By least loaded, we mean the **myth** machine that's running the fewest number of CS110 student processes.
    - Our **myth-buster** application is representative of the type of thing load balancers (e.g. **myth.stanford.edu**, **www.facebook.com**, or **www.netflix.com**) run to determine which internal server your request should forward to.
  - The overall architecture of the program looks like that below. We'll present various ways to implement **compileCS110ProcessCountMap.**

```
static const char *kCS110StudentIDsFile = "studentsunets.txt";
int main(int argc, char *argv[]) {
  unordered_set<string> cs110Students;
  readStudentFile(cs110Students, argv[1] != NULL ? argv[1] : kCS110StudentIDsFile);
  map<int, int> processCountMap;
  compileCS110ProcessCountMap(cs110Students, processCountMap);
  publishLeastLoadedMachineInfo(processCountMap);
  return 0;
}
```

Lecture 12: Multithreading and Networking

- Implementing **myth-buster**!

```
static const char *kCS110StudentIDsFile = "studentsunets.txt";
int main(int argc, char *argv[]) {
  unordered_set<string> cs110Students;
  readStudentFile(cs110Students, argv[1] != NULL ? argv[1] : kCS110StudentIDsFile);
  map<int, int> processCountMap;
  compileCS110ProcessCountMap(cs110Students, processCountMap);
  publishLeastLoadedMachineInfo(processCountMap);
  return 0;
}
```

  - **readStudentFile** updates **cs110Students** to house the SUNet IDs of all students currently enrolled in CS110. There's nothing interesting about its implementation, so I don't even show it (though you can see its implementation right here).
  - **compileCS110ProcessCountMap** is more interesting, since it uses networking—our first networking example!—to poll all 16 **myth**s and count CS110 student processes.
  - **processCountMap** is updated to map **myth** numbers (e.g. 61) to process counts (e.g. 9).
  - **publishLeastLoadedMachineInfo** traverses **processCountMap** and and identifies the least loaded **myth**.

Lecture 12: Multithreading and Networking

- The networking details are hidden and packaged in a library routine with this prototype:

```
int getNumProcesses(int num, const unordered_set<std::string>& sunetIDs);
```

- **num** is the myth number (e.g. 54 for **myth54**) and **sunetIDs** is a hashset housing the SUNet IDs of all students currently enrolled in CS110 (according to our **/usr/class/cs110/repos/assign4** directory).
- Here is the sequential implementation of a **compileCS110ProcessCountMap**, which is very brute force and CS106B-ish:

```
static const int kMinMythMachine = 51;
static const int kMaxMythMachine = 66;
static void compileCS110ProcessCountMap(const unordered_set<string>& sunetIDs,
                                        map<int, int>& processCountMap) {
  for (int num = kMinMythMachine; num <= kMaxMythMachine; num++) {
    int numProcesses = getNumProcesses(num, sunetIDs);
    if (numProcesses >= 0) {
      processCountMap[num] = numProcesses;
      cout << "myth" << num << " has this many CS110-student processes: " << numProcesses <<
    }
  }
}
```

Lecture 12: Multithreading and Networking

- Here are two sample runs of **myth-buster-sequential**, which polls each of the **myth**s in sequence (i.e. without concurrency).

```
poohbear@myth61$ time ./myth-buster-sequential
myth51 has this many CS110-student processes: 62
myth52 has this many CS110-student processes: 133
myth53 has this many CS110-student processes: 116
myth54 has this many CS110-student processes: 90
myth55 has this many CS110-student processes: 117
myth56 has this many CS110-student processes: 64
myth57 has this many CS110-student processes: 73
myth58 has this many CS110-student processes: 92
myth59 has this many CS110-student processes: 109
myth60 has this many CS110-student processes: 145
myth61 has this many CS110-student processes: 106
myth62 has this many CS110-student processes: 126
myth63 has this many CS110-student processes: 317
myth64 has this many CS110-student processes: 119
myth65 has this many CS110-student processes: 150
myth66 has this many CS110-student processes: 133
Machine least loaded by CS110 students: myth51
Number of CS110 processes on least loaded machine: 62
poohbear@myth61$
```

```
poohbear@myth61$ time ./myth-buster-sequential
myth51 has this many CS110-student processes: 59
myth52 has this many CS110-student processes: 135
myth53 has this many CS110-student processes: 112
myth54 has this many CS110-student processes: 89
myth55 has this many CS110-student processes: 107
myth56 has this many CS110-student processes: 58
myth57 has this many CS110-student processes: 70
myth58 has this many CS110-student processes: 93
myth59 has this many CS110-student processes: 107
myth60 has this many CS110-student processes: 145
myth61 has this many CS110-student processes: 105
myth62 has this many CS110-student processes: 126
myth63 has this many CS110-student processes: 314
myth64 has this many CS110-student processes: 119
myth65 has this many CS110-student processes: 156
myth66 has this many CS110-student processes: 144
Machine least loaded by CS110 students: myth56
Number of CS110 processes on least loaded machine: 58
poohbear@myth61$
```

- Each call to **getNumProcesses** is slow (about half a second), so 16 calls adds up to about 16 times that. Each of the two runs took about 5 seconds.

Lecture 12: Multithreading and Networking

- Each call to **getNumProcesses** spends most of its time off the CPU, waiting for a network connection to be established.
- Idea: poll each **myth** machine in its own thread of execution. By doing so, we'd align the dead times of each **getNumProcesses** call, and the total execution time will plummet.

```
static void countCS110Processes(int num, const unordered_set<string>& sunetIDs,
                                map<int, int>& processCountMap, mutex& processCountMapLock,
                                semaphore& permits) {
  int count = getNumProcesses(num, sunetIDs);
  if (count >= 0) {
    lock_guard<mutex> lg(processCountMapLock);
    processCountMap[num] = count;
    cout << "myth" << num << " has this many CS110-student processes: " << count << endl;
  }
  permits.signal(on_thread_exit);
}

static void compileCS110ProcessCountMap(const unordered_set<string> sunetIDs,
                                        map<int, int>& processCountMap) {
  vector<thread> threads;
  mutex processCountMapLock;
  semaphore permits(8); // limit the number of threads to the number of CPUs
  for (int num = kMinMythMachine; num <= kMaxMythMachine; num++) {
    permits.wait();
    threads.push_back(thread(countCS110Processes, num, ref(sunetIDs),
                             ref(processCountMap), ref(processCountMapLock), ref(permits)));
  }
  for (thread& t: threads) t.join();
}
```

Lecture 12: Multithreading and Networking

- Here are key observations about the code on the prior slide:
  - Polling the **myth**s concurrently means updating **processCountMap** concurrently. That means we need a **mutex** to guard access to **processCountMap**.
  - The implementation of **compileCS110ProcessCountMap** wraps a **thread** around each call to **getNumProcesses** while introducing a **semaphore** to limit the number of threads to a reasonably small number.
  - Note we use an overloaded version of **signal**. This one accepts the **on_thread_exit** tag as its only argument.
    - Rather than signaling the **semaphore** right there, this version schedules the **signal** to be sent after the entire thread routine has exited, as the **thread** is being destroyed.
    - That's the correct time to really **signal** if you're using the **semaphore** to track the number of active threads.
  - This new version, called **myth-buster-concurrent**, runs in about 0.75 seconds. That's a substantial improvement.
  - The full implementation of **myth-buster-concurrent** sits right here.