Lecture 13: More Threading Examples: Mythbuster and Ice Cream Store

Principles of Computer Systems

Winter 2020

Stanford University

Computer Science Department

Lecturers: Chris Gregg and Nick Troccoli

PDF of this presentation

Threads

Introduction to Threads → Threads and Mutexes → Condition Variables and Semaphores → Multithreading Patterns → Multithreading Examples

Introduction to Threads	Threads and Mutexes	Condition Variables and Semaphores	Multithreading Patterns	Multithreading Examples
2/3	2/5	2/10	2/12	2/19

Today

- Mythbuster
- Midterm Comments
- Ice Cream Store Example

Lecture 13: Mythbuster

- 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 13: Mythbuster

- 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 13: Mythbuster

- 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 << endl;
    }
  }
}
```

Lecture 13: Mythbuster

- Here are two sample runs of **myth-buster-sequential**, which polls each of the **myths** 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 13: Mythbuster

- 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.

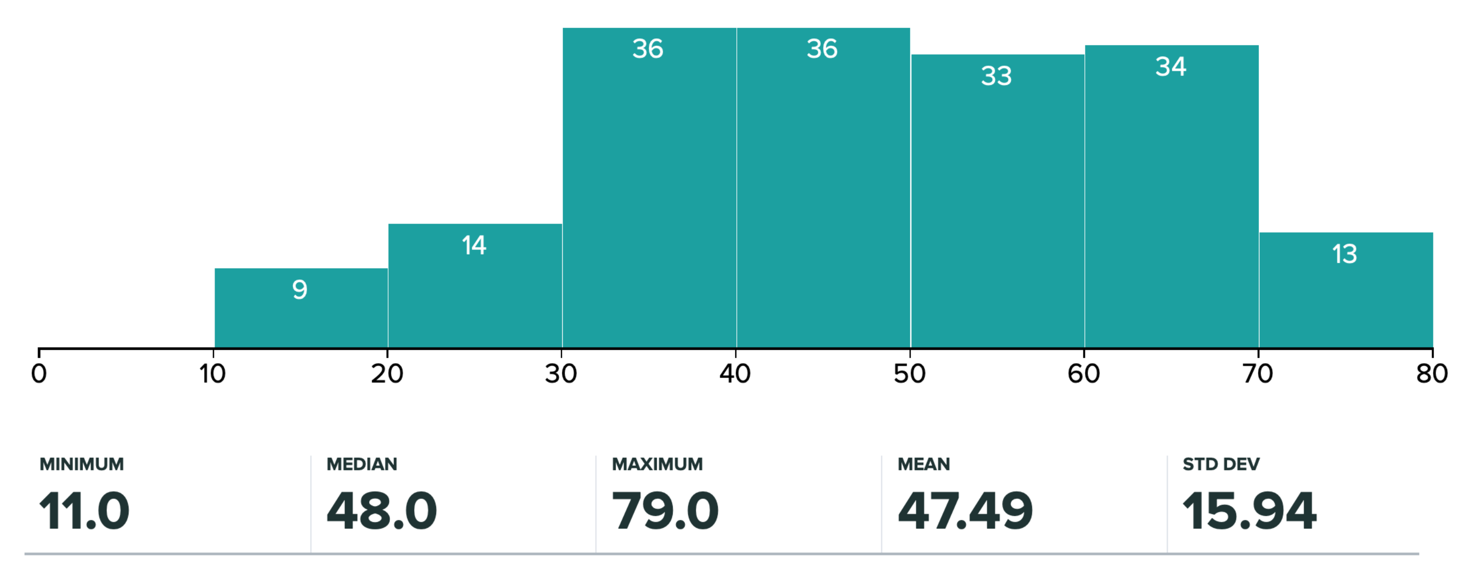
```
static void countCS110Processes(int num, const unordered_set<string>& sunetIDs,
                                map<int, int>& processCountMap, mutex& processCountMapLock,
                                semaphore& permits) {
  int count = getNumProcesses(num, sunetIDs);
  if (count >= 0) {
    lock_guard<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 13: Mythbuster

- 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.

Lecture 13: Midterm Comments



- The midterm median was a bit lower than expected (60%), but the overall distribution was about as expected otherwise.
- **`fork-chat`** was the most difficult problem
- **Signal mystery** was bimodal -- many students did not realize that a successful **`execvp`** never returns :(
- Regrade requests will open tomorrow
  - You must make your code work before we will regrade your original submission (Why? Because if it merits regrading, it should be possible to make it work correctly with some minor changes)

Lecture 13: An Ice Cream Store

- We will finish up today with a rather involved program that demonstrates how **`thread`**s can communicate with each other through the use of **`mutex`**es and **`semaphore`**s (which, as you will recall, is based on a **`conditional_variable_any`**).
- We are going to discuss five primary ideas:
  1. The *binary lock*
  2. A *generalized counter*
  3. A *binary rendezvous*
  4. A *generalized rendezvous*
  5. *Layered construction*
- Using these ideas, we will construct an ice cream store simulation that involves customers, clerks, managers, and cashiers. The original program was created as a final exam question by Julie Zelenski in CS 107 when CS 107 also taught multithreading.
- There is a handout with all of the code (on paper in class), which you can download here.
- You can download the runnable code here.

Lecture 13: An Ice Cream Store, idea 1: the binary lock

- We have already discussed binary locks in some detail
  - Using a **`mutex`**, we construct a *single-owner lock*.
  - When created, a **`mutex`** is unlocked and brackets *critical regions* of code that are matched with **`lock`** and **`unlock`** calls on the **`mutex`**. (We can also use a **`lock_guard<mutex>`** if the situation calls for it and our lock can go out of scope without further non-locked code being run).
  - This concurrency pattern locks down sole access to some shared state or resource that only one thread can be manipulating at a time.

Lecture 13: An Ice Cream Store, idea 2: a generalized counter

- When we use a **`semaphore`**, we can track the use of a resource, be it empty buffers, full buffers, available network connection, or what have you.
- The **`semaphore`** is essentially an integer count, capitalizing on its atomic increment, decrement, and the efficient blocking when a decrement is levied against a zero.
- A **`semaphore`** is constructed to the initial count on the resource (sometimes 0, sometimes N—it depends on the situation).
- As threads require a resource, they **`wait`** on the **`semaphore`** to transactionally consume it.
- Other threads (possibly, but not necessarily the same threads that consume) **`signal`** the **`semaphore`** when a new resource becomes available.
- This sort of pattern is used to efficiently coordinate shared use of a limited resource that has a *discrete* quantity. It can also be used to limit throughput (such as in the Dining Philosophers problem) where unmitigated contention might otherwise lead to deadlock.

Lecture 13: An Ice Cream Store, idea 3: a binary rendezvous

- When we use a **`semaphore`**, we can coordinate cross-thread communication.
- Suppose **`thread`** A needs to know when **thread** B has finished some task before it itself can progress any further.
- Rather than having A repeatedly loop (e.g. busy wait) and check some global state, a ***binary rendezvous*** can be used to foster communication between the two.
- The rendezvous semaphore is initialized to 0. When **`thread`** A gets to the point that it needs to know that another **`thread`** has made enough progress, it can **`wait`** on the rendezvous **`semaphore`**.
- After completing the necessary task, B will **`signal`** it. If A gets to the rendezvous point before B finishes the task, it will efficiently block until B's **`signal`**. If B finishes the task first, it **`signal`**s the semaphore, recording that the task is done, and when A gets to the **`wait`**, it will be sail right through it.
- A binary rendezvous **`semaphore`** records the status of one event and only ever takes on the value 0 (not-yet-completed or completed-and-checked) and 1 (completed-but-not-yet-checked).
- This concurrency pattern is sometimes used to wakeup another **`thread`** (such as disk reading **`thread`** that should spring into action when a request comes in), or to coordinate two dependent actions (a print job request that can't complete until the paper is refilled), and so forth.
- If you need a bidirectional rendezvous where both **`thread`**s need to wait for the other, you can add another **`semaphore`** in the reverse direction (e.g. the **`wait`** and **`signal`** calls inverted).
- Be careful that both **`thread`**s don’t try to **`wait`** for the other first and **`signal`** afterwards, else you can quickly arrive at deadlock!

Lecture 13: An Ice Cream Store, idea 4: a generalized rendezvous

- The ***generalized rendezvous*** is a combination of binary rendezvous and generalized counter, where a single `semaphore` is used to record how many times something has occurred.
- For example, if `thread` A spawned 5 `thread` Bs and needs to wait for all of them make a certain amount of progress before advancing, a generalized rendezvous might be used.
- The generalized rendezvous is initialized to 0. When A needs to sync up with the others, it will call `wait` on the `semaphore` in a loop, one time for each `thread` it is syncing up with. A doesn't care which specific thread of the group has finished, just that another has. If A gets to the rendezvous point before the `thread`s have finished, it will block, waking to "count" each child as it `signal`s and eventually move on when all dependent `thread`s have checked back.
- If all the B `thread`s finish before A arrives at the rendezvous point, it will quickly decrement the multiply-incremented `semaphore`, once for each `thread`, and move on without blocking.
- The current value of the generalized rendezvous `semaphore` gives you a count of the number of tasks that have completed that haven't yet been checked, and it will be somewhere between 0 and N at all times.
- The generalized rendezvous pattern is most often used to regroup after some divided task, such as waiting for several network requests to complete, or blocking until all pages in a print job have been printed.
- As with the generalized counter, it’s occasionally possible to use `thread::join` instead of `semaphore::wait`, but that requires the child `thread`s fully exit before the `join`ing parent is notified, and that’s not always what you want (though if it is, then `join` is just fine).

Lecture 13: An Ice Cream Store, idea 5: layered construction

- Once you have the basic patterns down, you can start to think about how `mutex`es and `semaphore`s can be *layered* and grouped into more complex constructions.
- Consider, for example, the constrained dining philosopher solution in which a generalized counter is used to limit throughput and `mutex`es are used for each of the forks.
- Another layered construct might be a global integer counter with a `mutex` lock and a binary rendezvous that can do something similar to that of a generalized rendezvous. As tasks complete, they can each lock and decrement the global counter, and when the counter gets to 0, a single `signal` to a rendezvous point can be sent by the last thread to finish.
- The combination of `mutex` and binary rendezvous `semaphore` could be used to set up a "race": `thread` C `wait`s for the first of `thread`s A and B to `signal`. `thread`s A and B each compete to be one who `signal`s the rendezvous. `thread` C only expects exactly one `signal`, so the `mutex` is used to provide critical-region access so that only the first thread `signal`s, but not the second.

Lecture 13: An Ice Cream Store: the simulation

- The program we will create simulates the daily activity in an ice cream store.
- The simulation's actors are the **clerks** who make ice cream cones, the single **manager** who supervises, the **customers** who buy ice cream cones, and the single **cashier** who accepts payment from customers. A different thread is launched for each of these actors.
- Each `customer` orders a few ice cream cones, waits for them to be made, gets in line to pay, and then leaves.
- `customer`s are in a big hurry and don't want to wait for one `clerk` to make several cones, so each `customer` dispatches one `clerk` thread for each ice cream cone he/she orders.
- Once the `customer` has all ordered ice cream cones, he/she gets in line at the `cashier` and waits his/her turn. After paying, each `customer` leaves.

Lecture 13: An Ice Cream Store: the simulation (continued)

- Each `clerk` thread makes exactly one ice cream cone. The `clerk` scoops up a cone and then has the `manager` take a look to make sure it is absolutely perfect. If the cone doesn't pass muster, it is thrown away and the `clerk` makes another. Once an ice cream cone is approved, the `clerk` hands the gem of an ice cream cone to the `customer` and is then done.
- The single `manager` sits idle until a clerk needs his or her freshly scooped ice cream cone inspected. When the `manager` hears of a request for an inspection, he/she determines if it passes and lets the `clerk` know how the cone fared. The `manager` is done when all cones have been approved.
- The `customer` checkout line must be maintained in FIFO order. After getting their cones, a customer "takes a number" to mark their place in the `cashier` queue. The `cashier` always processes `customer`s from the queue in order.
- The `cashier` naps while there are no `customer`s in line. When a `customer` is ready to pay, the `cashier` handles the bill. Once the bill is paid, the `customer` can leave. The `cashier` should handle the `customer`s according to number. Once all `customer`s have paid, the `cashier` is finished and leaves.

Lecture 13: An Ice Cream Store: the simulation (continued)

- Let's look at the following in turn:
  - random time/cone-perfection generation functions
  - `struct inspection`
  - `struct checkout`
  - `customer`
    - `browse`
  - `clerk`
    - `makeCone`
  - `manager`
    - `inspectCone`
  - `cashier`
  - `main`

Lecture 13: An Ice Cream Store: random generation functions

- Because we are modeling a "real" ice cream store, we want to randomize the times for each event. We also want to generate a boolean that says yay/nay about whether a cone is perfect. The following functions accomplished this task:

```
static mutex rgenLock;
static RandomGenerator rgen;

static unsigned int getNumCones() {
  lock_guard<mutex> lg(rgenLock);
  return rgen.getNextInt(kMinConeOrder, kMaxConeOrder);
}

static unsigned int getBrowseTime() {
  lock_guard<mutex> lg(rgenLock);
  return rgen.getNextInt(kMinBrowseTime, kMaxBrowseTime);
}

static unsigned int getPrepTime() {
  lock_guard<mutex> lg(rgenLock);
  return rgen.getNextInt(kMinPrepTime, kMaxPrepTime);
}

static unsigned int getInspectionTime() {
  lock_guard<mutex> lg(rgenLock);
  return rgen.getNextInt(kMinInspectionTime, kMaxInspectionTime);
}

static bool getInspectionOutcome() {
  lock_guard<mutex> lg(rgenLock);
  return rgen.getNextBool(kConeApprovalProbability);
}
```

Lecture 13: An Ice Cream Store: struct inspection

- There are two global structs -- because threads share global address space, this is the easiest way to handle them. We could, of course, create the data structures in **`main`** and then pass them into each thread and function by reference or pointer, but this simplifies it (though it does pollute the global namespace).
- The first struct we will look at is the inspection struct:

```
1 struct inspection {
2   mutex available;
3   semaphore requested;
4   semaphore finished;
5   bool passed;
6 } inspection;
```

- This **`struct`** coordinates between the clerk and the manager.
- The **`available`** mutex ensures the manager's undivided attention, so the single manager can only inspect one cone cone at a time
- The **`requested`** and **`finished`** semaphores coordinate a bi-directional rendezvous between the clerk and the manager.
- The **`passed`** bool provides the approval for a single cone.
- Note that we declare a variable of the struct right after the definition (line 6). This is the global variable (the struct definition itself, while global too, does not take up memory).

Lecture 13: An Ice Cream Store: struct checkout

- The second struct we will look at is the checkout struct:

```
1 struct checkout {
2   checkout(): nextPlaceInLine(0) {}
3   atomic<unsigned int> nextPlaceInLine;
4   semaphore customers[kNumCustomers];
5   semaphore waitingCustomers;
6 } checkout;
```

- This **`struct`** coordinates between the customers and the cashier.
- The **`nextPlaceInLine`** variable is a new, *atomic* variable that guarantees that ++ and -- work correctly without any data races.
- The **`customers`** array-based queue of semaphores allows the cashier to tell the customers that they have paid.
- The **`waitingCustomers`** semaphore informs the cashier that there are customers waiting to pay.
- Again, we define the global variable **`checkout`** on line 6.

Lecture 13: An Ice Cream Store: customer function

- Customers in our ice cream store, order cones, browse while waiting for them to be made, then wait in line to pay, and then leave. The customer function handles all of the details of the customer's ice cream store visit:

```
static void customer(unsigned int id, unsigned int numConesWanted) {
  // order phase
  vector<thread> clerks;
  for (unsigned int i = 0; i < numConesWanted; i++)
    clerks.push_back(thread(clerk, i, id));
  browse();
  for (thread& t: clerks) t.join();

  // checkout phase
  int place;
  cout << oslock << "Customer " << id << " assumes position #"
       << (place = checkout.nextPlaceInLine++) << " at the checkout counter."
       << endl << osunlock;
  checkout.waitingCustomers.signal();
  checkout.customers[place].wait();
  cout << "Customer " << id << " has checked out and leaves the ice cream store."
       << endl << osunlock;
}
```

- The customer needs one clerk for each cone. The customer browses and then must `join` all of the threads before checking out.
- The customers line up by `signal`ing for checkout.
- The customers `wait` in line until they are checked out.
- Note that the customer starts a clerk thread, and clerks are not waiting around like the manager or cashier.

Lecture 13: An Ice Cream Store: browse function

- The browse function is straightforward:

```
static void browse() {
  cout << oslock << "Customer starts to kill time." << endl << osunlock;
  unsigned int browseTime = getBrowseTime();
  sleep_for(browseTime);
  cout << oslock << "Customer just killed " << double(browseTime)/1000
       << " seconds." << endl << osunlock;
}
```

- The `sleep_for` function pushes the thread off the processor, so it is not busy-waiting.

Lecture 13: An Ice Cream Store: clerk function

- A clerk has multiple duties: make a cone, then pass it to a manager and wait for it to be inspected, then check to see if the inspection passed, and if not, make another and repeat until a well-made cone passes inspection:

```
static void clerk(unsigned int coneID, unsigned int customerID) {
  bool success = false;
  while (!success) {
    makeCone(coneID, customerID);
    inspection.available.lock();
    inspection.requested.signal();
    inspection.finished.wait();
    success = inspection.passed;
    inspection.available.unlock();
  }
}
```

- The clerk and the manager use the **`inspection`** struct to pass information -- note that there is only a single **`inspection`** struct, but that is okay because there is only one manager doing the inspecting.
  - This does not mean that we can remove the **`available`** lock -- it is critical because there are many clerks trying to get the manager's attention.
- Note that we only acquire the lock after making the cone -- don't over-lock.
- Note also that we signal the manager that we have a cone ready for inspection -- this wakes up the manager if they are sleeping. If the manger is in the middle of an inspection, they will immediately go to the next cone after the inspection.

Lecture 13: An Ice Cream Store: makeCone function

- The makeCone function is straightforward:

```
static void makeCone(unsigned int coneID, unsigned int customerID) {
  cout << oslock << "    Clerk starts to make ice cream cone #" << coneID
       << " for customer #" << customerID << "." << endl << osunlock;
  unsigned int prepTime = getPrepTime();
  sleep_for(prepTime);
  cout << oslock << "    Clerk just spent " << double(prepTime)/1000
       << " seconds making ice cream cone#" << coneID
       << " for customer #" << customerID << "." << endl << osunlock;
}
```

Lecture 13: An Ice Cream Store: manager function

- The manager (somehow) starts out the day knowing how many cones they will have to approve (we could probably handle this with a global "all done!" flag)
- The manager waits around for a clerk to hand them a cone to inspect.For each cone that needs to be approved, the manager inspects the cone, then updates the number of cones approved (locally) if it passes. If it doesn't pass, the manger waits again. When the manager has passed all necessary cones, they go home.

```
static void manager(unsigned int numConesNeeded) {
  unsigned int numConesAttempted = 0; // local variables secret to the manager,
  unsigned int numConesApproved = 0;  // so no locks are needed
  while (numConesApproved < numConesNeeded) {
    inspection.requested.wait();
    inspectCone();
    inspection.finished.signal();
    numConesAttempted++;
    if (inspection.passed) numConesApproved++;
  }

  cout << oslock << "  Manager inspected a total of " << numConesAttempted
       << " ice cream cones before approving a total of " << numConesNeeded
       << "." << endl;
  cout << "  Manager leaves the ice cream store." << endl << osunlock;
}
```

- The manager signals the waiting clerk that the cone has been inspected (why can there only be one waiting clerk?)

Lecture 13: An Ice Cream Store: inspectCone function

- The inspectCone function updates the inspection struct:

```
static void inspectCone() {
  cout << oslock << "  Manager is presented with an ice cream cone."
       << endl << osunlock;
  unsigned int inspectionTime = getInspectionTime();
  sleep_for(inspectionTime);
  inspection.passed = getInspectionOutcome();
  const char *verb = inspection.passed ? "APPROVED" : "REJECTED";
  cout << oslock << "  Manager spent " << double(inspectionTime)/1000
       << " seconds analyzing presented ice cream cone and " << verb << " it."
       << endl << osunlock;
}
```

- Why aren't there any locks needed here? This is a global struct!

Lecture 13: An Ice Cream Store: cashier function

- The cashier (somehow) knows how many customers there will be during the day. Again, we could probably handle telling the cashier to go home with a global variable.
- The cashier first waits for a customer to enter the line, and then signals that particular customer that they have paid.

```
static void cashier() {
  cout << oslock << "      Cashier is ready to take customer money."
       << endl << osunlock;
  for (unsigned int i = 0; i < kNumCustomers; i++) {
    checkout.waitingCustomers.wait();
    cout << oslock << "      Cashier rings up customer " << i << "."
         << endl << osunlock;
    checkout.customers[i].signal();
  }
  cout << oslock << "      Cashier is all done and can go home." << endl;
}
```

- Could we have handled the customer/cashier as we handled the clerks/manager, without the array?

Lecture 13: An Ice Cream Store: cashier function

- The cashier (somehow) knows how many customers there will be during the day. Again, we could probably handle telling the cashier to go home with a global variable.
- The cashier first waits for a customer to enter the line, and then signals that particular customer that they have paid.

```
static void cashier() {
  cout << oslock << "      Cashier is ready to take customer money."
       << endl << osunlock;
  for (unsigned int i = 0; i < kNumCustomers; i++) {
    checkout.waitingCustomers.wait();
    cout << oslock << "      Cashier rings up customer " << i << "."
         << endl << osunlock;
    checkout.customers[i].signal();
  }
  cout << oslock << "      Cashier is all done and can go home." << endl;
}
```

- Could we have handled the customer/cashier as we handled the clerks/manager, without the array?
  - We must ensure that customers get handled in order (otherwise, *chaos*) -- not so for the clerks, who can fight for the manager's attention.

Lecture 13: An Ice Cream Store: main function

- Finally, we can look at the main function.
- The main function's job is to set up the customers, manager, and cashier. Why not the clerks? (they are set up in the customer function)

```
int main(int argc, const char *argv[]) {
  int totalConesOrdered = 0;
  thread customers[kNumCustomers];
  for (unsigned int i = 0; i < kNumCustomers; i++) {
    int numConesWanted = getNumCones();
    customers[i] = thread(customer, i, numConesWanted);
    totalConesOrdered += numConesWanted;
  }
  thread m(manager, totalConesOrdered);
  thread c(cashier);

  for (thread& customer: customers) customer.join();
  c.join();
  m.join();
  return 0;
}
```

- `main` must wait for all of the threads it created to join before exiting.
- Now we see how the manager and cashier know how many cones / customers there are -- we let everyone in at the beginning, ask them how many cones they want, and off we go.

Lecture 13: An Ice Cream Store: takeaways

- There is a lot going on in this program!
- Managing all of the threads, locking, waiting, etc., takes planing and foresight.
- This isn't the only way to model the ice cream store
  - How would you modify the model?
  - What would we have to do if we wanted more than one manager?
  - Could we create multiple clerks in main, as well? (sure)
- This example prepares us for the next idea: **`ThreadPool`**.
  - Our manager and cashier threads are just waiting around much of the time, but they are created before needing to do their work.
  - It does take time to spin up a thread, so if we have the threads already waiting, we can use them quickly. This is similar to **`farm`**, except that now, instead of processes, we have threads.