CS110 Lecture 19: Thread Pool and Ice Cream Store

CS110: Principles of Computer Systems

Winter 2021-2022

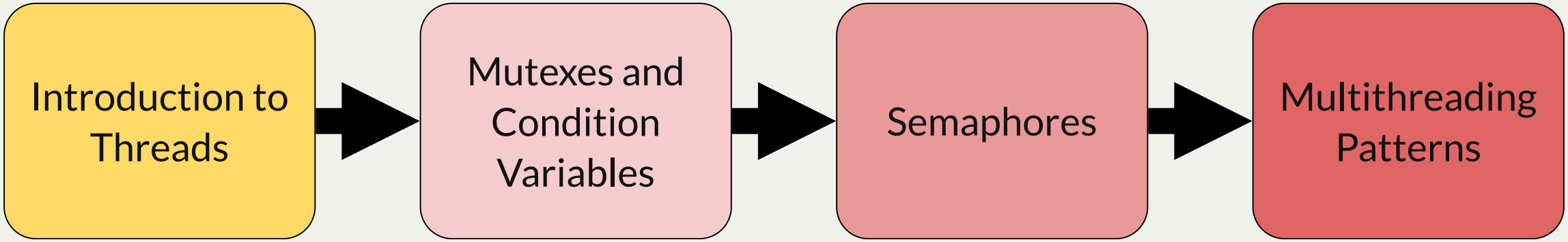
Stanford University

Instructors: Nick Troccoli and Jerry Cain

PDF of this presentation

CS110 Topic 3: How can we have concurrency within a single process?

Learning About Multithreading



Lecture 13 | **Lectures 14/15** | **Lecture 16** | **Lecture 17/18/this lecture**

assign5: implement your own multithreaded news aggregator to quickly fetch news from the web!

Learning Goals

- Practice applying our toolbox of concurrency directives (mutexes, condition variables and semaphores) to coordinate threads in different ways
- Understand the larger ice cream store example as a case study in multithreading and threads doing different tasks

Plan For Today

- **Recap:** Mythbuster
- **Example**: Ice Cream Store

Plan For Today

- **Recap: Mythbuster**
- **Example**: Ice Cream Store

Mythbuster

Let's implement a program called **myth-buster** that prints out how many CS110 student processes are running on each myth machine right now.

representative of load balancers (e.g. **myth.stanford.edu** or **www.netflix.com**) determining which internal server your request should forward to.

```
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
```

I/O-Bound vs. CPU-Bound Programs

CPU-bound tasks: the time to complete them is dictated by how long it takes us to do the CPU computation.

- heavy computations
- data processing

I/O-bound tasks: the time to complete them is dictated by how long it takes for some external mechanism to complete its work.

- reading from an external device (e.g. disk)
- reading data from the network

Even a single-core CPU can see performance improvements by parallelizing I/O-bound tasks. But parallelizing CPU-bound tasks will likely show minimal gains unless we have a multi-core CPU.

Parallelizing Mythbuster

For mythbuster, the primary task is fetching the number of running CS110 processes over the network. Is this an I/O-bound or CPU-bound task? **I/O-bound!**

This means we should see large gains from multithreading, even on a single-core machine.

Mythbusters: Concurrent

Implementation: spawn multiple threads, each responsible for connecting to a different myth machine and updating the map.

```
1  static void countCS110ProcessesForMyth(int mythNum, const unordered_set<string>& sunetIDs,
2    map<int, int>& processCountMap, mutex& processCountMapLock) {
3
4    int numProcesses = getNumProcesses(mythNum, sunetIDs);
5
6    // If successful, add to the map and print out
7    if (numProcesses >= 0) {
8      processCountMapLock.lock();
9      processCountMap[mythNum] = numProcesses;
10     processCountMapLock.unlock();
11     cout << oslock << "myth" << mythNum << " has this many CS110-student processes: " << numProcesses << endl << osunlock;
12   }
13 }
```

myth-buster-concurrent.cc

Mythbusters: Capped

When spawning threads, we don't want to spawn too many, because we might overwhelm the OS and diminish the performance gains of our multithreaded implementation.

A common approach is to **limit the number of simultaneous threads** with a *cap*. E.g. we can only have 16 spawned threads at a time. Once one finishes, then we can spawn another.

myth-buster-concurrent.cc

Mythbusters: Capped

- For each myth machine number, we'll spawn a new thread if there are permits available. That thread will fetch the count for that myth machine.
- When the thread finishes, it returns its permit.

```
1  static void createCS110ProcessCountMap(const unordered_set<string>& sunetIDs, map<int, int>& processCountMap) {
2    vector<thread> threads;
3    mutex processCountMapLock;
4    semaphore permits(kMaxNumSimultaneousThreads);
5
6    for (int mythNum = kMinMythMachine; mythNum <= kMaxMythMachine; mythNum++) {
7      permits.wait();
8
9      threads.push_back(thread(countCS110ProcessesForMyth, mythNum, ref(sunetIDs),
10       ref(processCountMap), ref(processCountMapLock), ref(permits)));
11   }
12
13   for (thread& threadToJoin : threads) threadToJoin.join();
14 }
```

myth-buster-concurrent.cc

Mythbusters: Capped

- For each myth machine number, we'll spawn a new thread if there are permits available. That thread will fetch the count for that myth machine.
- **When the thread finishes, it returns its permit.** We can use a special version of **signal()** to specify that the semaphore should be signaled only once it exits.

```
1  static void countCS110ProcessesForMyth(int mythNum, const unordered_set<string>& sunetIDs,
2    map<int, int>& processCountMap, mutex& processCountMapLock, semaphore& permits) {
3
4    permits.signal(on_thread_exit);
5
6    int numProcesses = getNumProcesses(mythNum, sunetIDs);
7
8    if (numProcesses >= 0) {
9      processCountMapLock.lock();
10     processCountMap[mythNum] = numProcesses;
11     processCountMapLock.unlock();
12     cout << "myth" << mythNum << " has this many CS110-student processes: " << numProcesses << endl;
13   }
14 }
```

myth-buster-concurrent.cc

Mythbusters: Thread Pool

Even though we are limiting the number of simultaneous threads, we still spawn that many in *total*. **It would be nice if we could use the *same* threads to complete all the tasks.**

A common approach is to use a **thread pool**; a variable type that maintains a pool of worker threads that can complete assigned tasks.

- You initialize the thread pool and specify the number of workers
- You can call **schedule** and pass in a function you want it to execute. It will assign it to the next available worker.
- You can call **wait** to block until all currently-assigned tasks have been completed.

```
class ThreadPool {
public:
   ThreadPool(size_t numThreads);
   void schedule(const std::function<void(void)>& thunk);
   void wait();
   ~ThreadPool();
};
```

Mythbusters: Thread Pool

Even though we are limiting the number of simultaneous threads, we still spawn that many in *total*. **It would be nice if we could use the *same* threads to complete all the tasks.**

What might this look like in code?

- In myth buster, instead of spawning threads, we can **schedule** a "thunk" for each task of fetching a myth machine's count of CS110 processes. **It must be a function that has no parameters or return value**.
- After we add all the tasks to the thread pool, we **wait** on the thread pool to finish all the tasks.

```
class ThreadPool {
public:
   ThreadPool(size_t numThreads);
   void schedule(const std::function<void(void)>& thunk);
   void wait();
   ~ThreadPool();
};
```

Mythbusters: Thread Pool

- We can schedule a "thunk" for each task of fetching a myth machine's count of CS110 processes. It must be a function that has no parameters or return value.
- After we add all the tasks to the thread pool, we **wait** on the thread pool to finish all the tasks.

```
1  static void createCS110ProcessCountMap(const unordered_set<string>& sunetIDs,
2            map<int, int>& processCountMap) {
3
4    ThreadPool pool(kMaxNumSimultaneousThreads);
5    mutex processCountMapLock;
6
7    for (int mythNum = kMinMythMachine; mythNum <= kMaxMythMachine; mythNum++) {
8      pool.schedule([mythNum, &sunetIDs, &processCountMap, &processCountMapLock]() {
9        countCS110ProcessesForMyth(mythNum, sunetIDs, processCountMap, processCountMapLock);
10     });
11   }
12   ...
```

We can only enqueue a task represented by a function with no params/return value. Therefore, to access external data, we must capture it in a lambda function.

myth-buster-pooled.cc

Mythbusters: Thread Pool

- We can schedule a "thunk" for each task of fetching a myth machine's count of CS110 processes. It must be a function that has no parameters or return value.
- **After we add all the tasks to the thread pool, we wait on the thread pool to finish all the tasks.**

```
1  static void createCS110ProcessCountMap(const unordered_set<string>& sunetIDs,
2            map<int, int>& processCountMap) {
3
4    ThreadPool pool(kMaxNumSimultaneousThreads);
5    mutex processCountMapLock;
6
7    for (int mythNum = kMinMythMachine; mythNum <= kMaxMythMachine; mythNum++) {
8      pool.schedule([mythNum, &sunetIDs, &processCountMap, &processCountMapLock]() {
9        countCS110ProcessesForMyth(mythNum, sunetIDs, processCountMap, processCountMapLock);
10     });
11   }
12
13   pool.wait();
14 }
```

myth-buster-pooled.cc

Thread Pools

Thread Pools are very useful abstractions that let a client spread tasks across several threads without having to deal with the complexities of threads.

- You will have a chance to implement your own **ThreadPool** class on assignment 5!

Plan For Today

- **Recap:** Mythbuster
- **Example: Ice Cream Store**

Visiting The Ice Cream Store

- Now, let's use our multithreading knowledge to understand an in-depth multithreading program simulating an ice cream store!
- There are **customers**, **clerks**, a **manager** and a **cashier**, coordinating in various ways.

ice-cream-store.cc

Visiting The Ice Cream Store

- Each **customer** wants to order some number of ice cream cones.
- A **customer** spawns a new **clerk** to make each ice cream cone.
- A **clerk** makes a single cone, and must have it approved by the **manager**.
- The single **manager** approves or rejects cones made by **clerks**.
- Once a **customer's** order is made, they must get in line with the **cashier** to check out.
- The **cashier** helps **customers** check out *in the order in which they got on line.*

https://stanford-pilot.hosted.panopto.com/Panopto/Pages/Embed.aspx?id=a2143571-264d-470c-bef8-ae3e0073964e&autoplay=false&offerviewer=false&showtitle=false&showbrand=true&captions=false&interactivity=all

Ice Cream Store: scaffolding

```
1  static mutex rgenLock;
2  static RandomGenerator rgen;
3
4  ...
5
6  void browse() {
7    cout << oslock << "Customer starts to kill time." << endl << osunlock;
8    size_t browseTimeMS = getBrowseTimeMS();
9    sleep_for(browseTimeMS);
10   cout << oslock << "Customer just killed " << double(browseTimeMS) / 1000
11        << " seconds." << endl << osunlock;
12 }
13
14 void makeCone(size_t coneID, size_t customerID) {
15   cout << oslock << "    Clerk starts to make ice cream cone #" << coneID
16        << " for customer #" << customerID << "." << endl << osunlock;
17   size_t prepTimeMS = getPrepTimeMS();
18   sleep_for(prepTimeMS);
19   cout << oslock << "    Clerk just spent " << double(prepTimeMS) / 1000
20        << " seconds making ice cream cone #" << coneID
21        << " for customer #" << customerID << "." << endl << osunlock;
22 }
23
24 ...
```

To model a "real" ice cream store, we want to randomize different occurrences throughout the program. We use functions like this to do that.

Ice Cream Store: main

```
1  int main(int argc, const char *argv[]) {
2    // Make an array of customer threads, and add up how many cones they order
3    size_t totalConesOrdered = 0;
4    thread customers[kNumCustomers];
5
6    /* The structs to package up variables needed for cone inspection and
7     * customer checkout
8     */
9    inspection_t inspection;
10   checkout_t checkout;
11
12   for (size_t i = 0; i < kNumCustomers; i++) {
13     // utility function, random (see ice-cream-store-utils.h)
14     size_t numConesWanted = getNumCones();
15     customers[i] = thread(customer, i, numConesWanted,
16       ref(inspection), ref(checkout));
17     totalConesOrdered += numConesWanted;
18   }
19
20   /* Make the manager and cashier threads to approve cones / checkout customers.
21    * Tell the manager how many cones will be ordered in total. */
22   thread managerThread(manager, totalConesOrdered, ref(inspection));
23   thread cashierThread(cashier, ref(checkout));
24
25   for (thread& customer: customers) customer.join();
26   cashierThread.join();
27   managerThread.join();
28
29   return 0;
30 }
```

In **main**, we spawn all of the customers, the manager (telling it the total number of cones ordered), and the cashier. Why not clerks? Each customer spawns its own clerks.

Then, we wait for the threads to finish.

Ice Cream Store: customer

A customer does the following:

1. spawns a clerk for each cone
2. browses and waits for clerks to finish
3. gets its number in checkout line
4. tells cashier we are ready to check out
5. waits for cashier to ring us up

- "gets its number in checkout line" - global counter, needs a binary lock
- "tells cashier we are ready to check out" - one generalized coordination semaphore
- "waits for cashier to ring us up" - binary coordination semaphore *per customer*

Ice Cream Store: customer

```
1 struct checkout_t {
2   atomic<size_t> nextPlaceInLine{0};
3   semaphore customers[kNumCustomers];
4   semaphore waitingCustomers;
5 };
```

Struct passed by reference to all customers and the cashier.

- **nextPlaceInLine** is a counter that is *automatically atomic for ++!*
- **waitingCustomers** is a generalized coordination semaphore that the cashier waits on
- **customers** stores a binary coordination semaphore per customer, customers wait on them

Ice Cream Store: customer

```
1  static void customer(size_t id, size_t numConesWanted,
2                       inspection_t& inspection, checkout_t& checkout) {
3    // Make a vector of clerk threads, one per cone to be ordered
4    vector<thread> clerks(numConesWanted);
5    for (size_t i = 0; i < clerks.size(); i++) {
6      clerks[i] = thread(clerk, i, id, ref(inspection));
7    }
8
9    // The customer browses for some amount of time, then joins the clerks.
10   browse();
11   for (thread& clerk: clerks) clerk.join();
12
13   size_t place = checkout.nextPlaceInLine++;
14   cout << oslock << "Customer " << id << " assumes position #" << place
15        << " at the checkout counter." << endl << osunlock;
16
17   // Tell the cashier that we are ready to check out
18   checkout.waitingCustomers.signal();
19
20   // Wait on our unique semaphore so we know when it is our turn
21   checkout.customers[place].wait();
22   cout << oslock << "Customer " << id
23        << " has checked out and leaves the ice cream store."
24        << endl << osunlock;
25 }
```

A customer does the following:

1) spawns a clerk for each cone
2) browses and waits for clerks
3) gets its place in checkout line
4) tells cashier it's there
5) waits for cashier to ring it up

```
1 struct checkout_t {
2   atomic<size_t> nextPlaceInLine{0};
3   semaphore customers[kNumCustomers];
4   semaphore waitingCustomers;
5 };
```

Ice Cream Store: clerk

A clerk does the following:

1. makes a cone
2. attempts to get exclusive access to the manager
3. tells the manager it needs approval
4. waits for the manager to decide whether to approve or reject
5. checks the manager's decision
6. forfeits exclusive access to the manager
7. if our cone was rejected, go to step 1

- "attempts to get exclusive access to the manager" - binary lock
- "tells the manager it needs approval" - binary coordination semaphore
- "waits for the manager to decide..." - binary coordination semaphore

Ice Cream Store: clerk

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

Struct passed by reference to all clerks and the manager.

- **available** is a lock that a clerk must hold in order to interact with the manager.
- **requested** is a binary coordination semaphore that the manager waits on
- **finished** is a binary coordination semaphore that a clerk waits on
- **passed** stores the result of the most recent inspection - only for lock-holder.

Ice Cream Store: clerk

```
1  static void clerk(size_t coneID, size_t customerID,
2       inspection_t& inspection) {
3
4    bool success = false;
5    while (!success) {
6      makeCone(coneID, customerID);
7
8      // We must be the only one requesting approval
9      inspection.available.lock();
10
11     // Let the manager know we are requesting approval
12     inspection.requested.signal();
13
14     // Wait for the manager to finish
15     inspection.finished.wait();
16
17     /* If the manager is finished, it has put
18      * its approval decision into "passed"
19      */
20     success = inspection.passed;
21
22     // We're done requesting approval, so unlock for someone else
23     inspection.available.unlock();
24   }
25 }
```

A clerk does the following:

1. makes a cone
2. gets exclusive manager access
3. tells the manager it needs approval
4. waits for the manager to decide
5. checks the manager's decision
6. forfeits manager access
7. if rejected, go to step 1

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

Ice Cream Store: manager

The single manager does the following while there are more cones needed:

1. waits for a clerk to request an inspection
2. inspects the cone and records decision to approve or not
3. tells the clerk that it is done
4. updates its cone counts
5. if more cones needed, go to step 1

- "waits for a clerk's cone to inspect" - binary coordination semaphore
- "tells the clerk that we are done" - binary coordination semaphore

Ice Cream Store: manager

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

Struct passed by reference to all clerks and the manager.

- **available** is a lock that a clerk must hold in order to interact with the manager.
- **requested** is a binary coordination semaphore that the manager waits on
- **finished** is a binary coordination semaphore that a clerk waits on
- **passed** stores the result of the most recent inspection - only for lock-holder.

Ice Cream Store: manager

```
1  static void manager(size_t numConesNeeded,
2       inspection_t& inspection) {
3
4    size_t numConesAttempted = 0;
5    size_t numConesApproved = 0;
6
7    while (numConesApproved < numConesNeeded) {
8      // Wait for someone to request an inspection
9      inspection.requested.wait();
10
11     inspection.passed = inspectCone();
12
13     // Let them know we have finished inspecting
14     inspection.finished.signal();
15
16     numConesAttempted++;
17     if (inspection.passed) numConesApproved++;
18   }
19
20   cout << oslock << "  Manager inspected a total of "
21        << numConesAttempted
22        << " ice cream cones before approving a total of "
23        << numConesNeeded
24        << "." << endl << "  Manager leaves the ice cream store."
25        << endl << osunlock;
26 }
```

The manager does the following while there are more cones needed:

1. waits for a clerk's cone to inspect
2. inspects the cone and records decision to approve or not.
3. tells the clerk that it is done.
4. updates its cone counts
5. if more cones needed, go to 1

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

Ice Cream Store: cashier

The single cashier does the following while there are more customers to ring up:

1. waits for a customer to be ready to check out
2. tells the i-th customer that it has checked out
3. if more customers to ring up, go to step 1

- "waits for a customer to be ready to check out" - generalized coordination semaphore
- "tells the i-th customer that it has checked out" - binary coordination semaphore *per customer*

Ice Cream Store: cashier

```
1 struct checkout_t {
2   atomic<size_t> nextPlaceInLine{0};
3   semaphore customers[kNumCustomers];
4   semaphore waitingCustomers;
5 };
```

Global struct shared by all customers and the cashier.

- **nextPlaceInLine** is a counter that is *automatically atomic for ++!*
- **waitingCustomers** is a generalized coordination semaphore that the cashier waits on
- **customers** stores a binary coordination semaphore per customer, customers wait on them

Ice Cream Store: cashier

```
1  static void cashier(checkout_t& checkout) {
2    cout << oslock
3         << "      Cashier is ready to help customers check out."
4         << endl << osunlock;
5
6    // We check out all customers
7    for (size_t i = 0; i < kNumCustomers; i++) {
8      // Wait for someone to let us know they are ready to check out
9      checkout.waitingCustomers.wait();
10     cout << oslock << "      Cashier rings up customer " << i << "."
11          << endl << osunlock;
12
13     // Let the ith customer know that they can leave.
14     checkout.customers[i].signal();
15   }
16
17   cout << oslock << "      Cashier is all done and can go home."
18        << endl << osunlock;
19 }
```

The cashier does the following while there are more customers to ring up:

1. waits for a customer to be ready to check out
2. tells the i-th customer that it has checked out
3. if more customers to ring up, go to step 1

```
1 struct checkout_t {
2   atomic<size_t> nextPlaceInLine{0};
3   semaphore customers[kNumCustomers];
4   semaphore waitingCustomers;
5 };
```

Ice Cream Store Takeaways

- There's a lot going on in this simulation!
- Managing all of the threads, locking, waiting, etc., takes planning 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)
- Example of different threads doing different tasks
- Layered construction - combination of multithreading patterns
- Role playing helps to visualize!

Multithreading Wrap-Up

- Multithreading allows one process to execute multiple tasks at the same time.
- We can spawn threads, which all share the same address space, and each of them can execute a function.
- Race conditions are common when accessing shared data
- We can use concurrency directives like **mutexes**, **condition variables** and **semaphores** to coordinate between threads and prevent race conditions.
- Depending on what tasks a program performs, it may see varying benefits from adding multithreading - eg. I/O-bound vs. CPU-bound tasks.

Recap

- **Recap:** Mythbuster
- **Example**: Ice Cream Store

Next time: Introduction to networking