Lecture 9: Multithreading Recap

Conceptual questions from the ring buffer example

See the code from the previous lecture.

- What happens if we initialize `dataWritten` to be more than zero?
  - The reader thread might read from the buffer before the writer has written to it.
- What happens if we accidentally omit the initial value for `dataRead`?
  - The program deadlocks. The writer thread is waiting for space to be made available in the buffer, but the reader thread is waiting for data to be written.
- What if we initialize `dataRead` to be less than 10?
  - The writer thread just goes slower. It effectively shrinks the size of our ring buffer.
- What if we initialize `dataRead` to be *more* than 10?
  - This introduces a race condition, where the writer thread might overwrite part of the buffer before the reader thread has read it.

Example: Cluster statistics

Let's write a program to SSH into every myth machine, get statistics about that machine, and then aggregate all collected statistics in order to see which processes are most popular across the myth cluster.

In `cluster-stats.cc`, I've written a `getProcessCounts` function that logs into a particular myth machine and returns an `unordered_map<string, int>` mapping processes to their counts. (This is essentially a histogram of processes running on that machine.) We can write a simple program to aggregate these maps across all myth machines, then sort the results to get the top 20 processes:

```
unordered_map<string, int> getProcessCounts(unsigned short mythNum);

int main(int argc, const char *argv[]) {
    unordered_map<string, int> counts;

    for (unsigned short i = 51; i <= 66; i++) {
```

```
        cout << oslock << "Connecting to myth" << i << "..." << endl << osunlock;
        unordered_map<string, int> localCounts = getProcessCounts(i);
        for (auto p : localCounts) {
            counts[p.first] += p.second;
        }
    }

    vector<pair<string, int>> finalCounts;
    for (auto p : counts) {
        finalCounts.push_back(p);
    }
    sort(finalCounts.begin(), finalCounts.end(), [](pair<string, int> x, pair<stri
        return x.second > y.second;
    });

    cout << "Total counts:" << endl;
    for (size_t i = 0; i < 20; i++) {
        auto p = finalCounts[i];
        cout << p.second << "\t" << p.first << endl;
    }
}
```

Output:

```
Connecting to myth51...
Connecting to myth52...
Connecting to myth53...
Connecting to myth54...
Connecting to myth55...
Connecting to myth56...
Connecting to myth57...
Connecting to myth58...
Connecting to myth59...
Connecting to myth60...
Connecting to myth61...
Connecting to myth62...
Connecting to myth63...
Connecting to myth64...
```

```
Connecting to myth65...
Connecting to myth66...
Total counts:
5106    python ./factor.py --self-halting
200     (sd-pam)
200     /lib/systemd/systemd --user
160     [bioset]
98      wc
69      -bash
64      [afs_background]
54      /bin/bash
32      ./stsh
32      [hci0]
29      zsh
27      SCREEN
23      sshd: rebs [priv]
22      tmux
18      /usr/bin/sort
16      /usr/lib/x86_64-linux-gnu/indicator-messages/indicator-messages-service
16      [vmstat]
16      ps ax -o args
16      /usr/lib/bluetooth/bluetoothd
16      [scsi_eh_0]
```

This works! However, it took 21 seconds to run. We can do better using multithreading. We need a mutex in order to prevent concurrent updates to `counts`, but otherwise, the code is mostly the same.

```
unordered_map<string, int> counts;
mutex countsLock;
vector<thread> threads;

for (unsigned short i = 51; i <= 66; i++) {
    threads.push_back(thread([i, &counts, &countsLock](){
        cout << oslock << "Connecting to myth" << i << "..." << endl << osunlock;
        unordered_map<string, int> localCounts = getProcessCounts(i);
        lock_guard<mutex> lg(countsLock);
        for (auto p : localCounts) {
```

```
            counts[p.first] += p.second;
        }
    }));
}
for (thread &t : threads) {
    t.join();
}
```

This spawns one thread for each myth machine. That's okay for the small myth cluster, but what if we wanted to run this at Google? We'd spawn so many threads, it would overwhelm the scheduler.

We can use a semaphore to limit the number of threads in use at any particular time:

```
    unordered_map<string, int> counts;
    mutex countsLock;
    vector<thread> threads;
    semaphore threadSema(16);

    for (unsigned short i = 51; i <= 66; i++) {
        threadSema.wait();
        threads.push_back(thread([i, &counts, &countsLock, &threadSema](){
            cout << oslock << "Connecting to myth" << i << "..." << endl << osunlo
            unordered_map<string, int> localCounts = getProcessCounts(i);
            lock_guard<mutex> lg(countsLock);
            for (auto p : localCounts) {
                counts[p.first] += p.second;
            }
            threadSema.signal();
        }));
    }
```

This works, but there is a (very small) chance that we might end up with more than 16 worker threads if the threads get pulled off the processor right after the `signal` but right before actually exiting. We can address this by using a different variant of `semaphore.signal()`:

```
semaphore.signal(on_thread_exit)
```

This doesn't `signal` immediately, but instead schedules a signal to occur once the thread exits. This is *also* helpful because it guarantees the semaphore will be signaled, even if an exception occurs or we forget to signal the semaphore for some other reason.

```
// ...
threads.push_back(thread([i, &counts, &countsLock, &threadSema](){
    threadSema.signal(on_thread_exit);
    // ...
```

Example: Ice Cream Parlor

This example is a bit contrived, but it allows us to demonstrate a complex synchronization setup without actually having a very complicated program. It also introduces some inter-thread communication patterns that will be helpful in Assignment 4.

We are simulating an ice cream parlor, but one from an odd universe...

- There are 10 customers. Each decide to order 1-4 ice cream cones. However, instead of ordering them at a counter like they normally would, they hire clerks *on demand* to make their cones. If a customer comes in wanting 3 ice cream cones, he hires 3 clerks on the spot to make each cone for him.
- A single manager sits in an office. When clerks make the ice cream cones, he inspects them one-by-one and decides whether to approve or reject them.
- Each clerk that is hired rushes to the parlor and makes a cone, before presenting it to a manager for approval. If the manager approves the cone, the clerk hands it to the customer and leaves; otherwise, the clerk re-makes the cone.
- A single cashier rings up each customer's order.

Pseudocode for each function looks something like this:

```
int main(int argc, const char *argv[]) {
    vector<thread> customers;
    size_t totalCones = 0;
    for (size_t i = 0; i < 10; i++) {
        size_t count = randomInt(1, 4);
        customers.push_back(thread(customer, i, count));
        totalCones += count;
    }
    thread m(manager, totalCones);
```

```
    thread c(cashier);

    for (size_t i = 0; i < 10; i++) {
        customers[i].join();
    }
    m.join();
    c.join();
    return 0;
}

static void customer(size_t id, size_t coneCount) {
    // Ordering ice cream cones:

    // - create clerk threads, one for each cone
    // - join on all the clerk threads

    // Checkout:

    // - get in line to see the cashier
    // - wait until the cashier sees us
}

static void clerk(size_t customerId, size_t coneId) {
    // - while we haven't passed inspection:
    //     - get the manager's attention
    //     - present the ice cream cone
    //     - wait for the manager to approve or reject
    //     - leave the manager's office
}

static void manager(size_t numNeeded) {
    // - while we are expecting clerks to make cones:
    //     - wait for a clerk to enter the office
    //     - inspect the cone
    //     - notify the clerk of our decision
}

static void cashier() {
```

```
    // - while there are still customers that haven't checked out:
    //     - wait for the next customer to come to the line
    //     - ring up that customer
    //     - tell the customer we're done ringing them up
}
```

To implement this code, we use two global structs that hand off information between threads. (As usual, I'm using globals to simplify lecture, but you should use private instance variables in assignments.) One struct acts like the manager's "office," and the other struct acts like the cashier's line.

```
struct {
    mutex vacant;   // A clerk must acquire this lock to enter the office
    bool passedInspection;
    semaphore inspectionRequested;  // wake up the manager
    semaphore inspectionFinished;   // wake up the clerk after the inspection
} managerOffice;

struct {
    semaphore numWaiting;   // wake up the cashier when someone joins the line
    semaphore done[10];     // wake up a customer in line when they are ringed up
    size_t nextIndex = 0;   // when the next customer joins the line, they'll be
                            // in this position in line
    mutex nextIndexLock;
} cashierLine;

static void customer(size_t id, size_t coneCount) {
    // Order:
    sleep_for(randomTime());
    vector<thread> clerks;
    for (size_t i = 0; i < coneCount; i++) {
        clerks.push_back(thread(clerk, id, i));
    }
    for (size_t i = 0; i < coneCount; i++) {
        clerks[i].join();
    }
    // Checkout:
    cashierLine.nextIndexLock.lock();
```

```
    size_t place = cashierLine.nextIndex++;
    cashierLine.nextIndexLock.unlock();
    cout << oslock << "Customer " << id << " is in position #" << place
        << " at the checkout counter." << endl << osunlock;
    cashierLine.numWaiting.signal();
    cashierLine.done[place].wait();
    cout << oslock << "Customer " << id << " exits the ice cream store."
        << endl << osunlock;
}

static void clerk(size_t customerId, size_t coneId) {
    bool passedInspection = false;
    while (!passedInspection) {
        // Make cone
        cout << oslock << "Clerk making cone for customer " << customerId
            << endl << osunlock;
        sleep_for(randomTime());

        managerOffice.vacant.lock();
        managerOffice.inspectionRequested.signal();
        managerOffice.inspectionFinished.wait();
        passedInspection = managerOffice.passedInspection;
        managerOffice.vacant.unlock();
    }
}

static void manager(size_t numNeeded) {
    size_t numAttempted = 0;
    size_t numApproved = 0;
    while (numApproved < numNeeded) {
        managerOffice.inspectionRequested.wait();
        cout << oslock << "Manager has been presented an ice cream cone."
            << endl << osunlock;
        sleep_for(randomTime());
        managerOffice.passedInspection = randomChance(0.3);
        cout << oslock << "Manager "
            << (managerOffice.passedInspection ? "approved" : "rejected")
            << " ice cream cone." << endl << osunlock;
```

```
        managerOffice.inspectionFinished.signal();
        numAttempted++;
        if (managerOffice.passedInspection) numApproved++;
    }
    cout << oslock << "Manager has finished all inspections! He inspected "
        << numAttempted << " cones before approving a total of "
        << numApproved << " cones." << endl << osunlock;
}

static void cashier() {
    cout << oslock << "Cashier ready for business!" << endl << osunlock;
    for (size_t i = 0; i < 10; i++) {
        cashierLine.numWaiting.wait();
        cout << oslock << "Cashier rings up customer " << i << "."
            << endl << osunlock;
        cashierLine.done[i].signal();
    }
    cout << oslock << "Cashier heading home." << endl << osunlock;
}
```