Multithreading and Synchronization Redux

Based on a handout written by Julie Zelenski.

Concurrency Patterns

There's value in learning programming patterns—patterns such as linear search, or divide-and-conquer—that can be generalized and used in other programs. We at CS110 headquarters thought it might be helpful to describe the common concurrency patterns managed by the `thread`, the `mutex`, and our custom `semaphore` (and, yes, the `conditional_variable_any`, though I introduced them during lecture primarily so you understand how the `semaphore` is implemented). Hopefully, these descriptions can help you identify some of the ways `mutex`**e**s and `semaphore`s are used and give you some insight into what options you have for solving different concurrency and threading problems.

Binary lock: The `mutex` is used as a single-owner lock. When constructed, it is in an unlocked state and threads bracket critical regions with matched `lock` and `unlock` calls on the `mutex`. This sort of concurrency pattern is used to lock down sole access to some shared state or resource that only one thread can be manipulating at any one time.

Generalized counter: A `semaphore` is used to track some 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. The 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.

Binary rendezvous: The `semaphore` can be used to 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 communi-

cation 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!

Generalized rendezvous: 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).

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.

The Ice Cream Store Simulation

Before we move on to our next topic (networking: woot), here is one final executable that's a bit more involved than the simpler, more focused examples we've seen over the past 1.5 weeks. This problem is loosely based on a merged version of two old final exam questions given a long, long time ago when CS107 used to teach concurrency.

This program 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.

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.

We'll walk through the core of this problem and its solution together during lecture.

```
    /**
     * File: ice-cream-store.c
     * ----------------------
     * An larger example used to show a combination of synchronization
techniques
     * (binary locks, generalized counters, rendezvous semaphores) in a more
     * complicated arrangement.
     *
     * This is the ''ice cream store'' simulation. There are customers who
want
     * to buy ice cream cones, clerks who make the cones, the manager who
     * checks a clerk's work, and the cashier who takes a customer's
     * money. There are a many different interactions that need to be
modeled:
     * the customers dispatching several clerks (one for each cone they are
buying),
     * the clerks who need the manager to approve their work, the cashier
     * who rings up each customer in a first-in-first-out, and so on.
```

```
 */

#include <thread>
#include <mutex>
#include <vector>
#include <iostream>
#include <atomic>
#include ''semaphore.h''
#include ''ostreamlock.h''
#include ''random-generator.h''
#include ''thread-utils.h''
using namespace std;

static const unsigned int kNumCustomers = 5;
static const unsigned int kMinConeOrder = 1;
static const unsigned int kMaxConeOrder = 3;
static const unsigned int kMinBrowseTime = 100;
static const unsigned int kMaxBrowseTime = 300;
static const unsigned int kMinPrepTime = 50;
static const unsigned int kMaxPrepTime = 300;
static const unsigned int kMinInspectionTime = 20;
static const unsigned int kMaxInspectionTime = 100;
static const double kConeApprovalProbability = 0.8;

/**
 * Everything from here down to the next comment is used to
 * generate random sleep times and model the variations in
 * execution that come with a real, concurrent, mostly unpredictable
system.
 */
```

```
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);
}

/**
 * The inspection record is a single, global record used to
 * coordinate interaction between a single clerk and the manager.
 * The available mutex is used by a clerk to transactionally
```

```
 * acquire the one manager's undivided attention. The requested
 * and finished semaphores are used to coordinate bidirectional
 * rendezvous between the clerk and the manager. The passed
 * bool stores the approval status of the ice cream cone made by
 * the clerk holding the manager's attention.
 */

struct inspection {
  mutex available;
  semaphore requested;
  semaphore finished;
  bool passed;
} inspection;

/**
 * The checkout record is a single, global record used to
 * coordinate interactions between all of the customers and
 * the cashier. We introduce the atomic<unsigned int> as means
 * for providing an exposed nonnegative integer that supports
 * ++ and -- in such a way that it's guaranteed to be atomic
 * on all platforms. The waitingCustomers semaphore is used to
 * inform the cashier that one or more people are waiting in
 * line, and the array-based queue of semaphores are used to foster
 * cashier-to-customer rendezvous so each customer knows that
 * his/her payment has been accepted.
 */

struct checkout {
    checkout(): nextPlaceInLine(0) {}
    atomic<unsigned int> nextPlaceInLine;
    semaphore customers[kNumCustomers];
```

```
        semaphore waitingCustomers;
    } checkout;

    /**
     * Utility functions: browse and makeCone
     * --------------------------------------
     * browse and makeCone are called by customers and
     * clerks to stall and/or emulate the time it might take
     * to do something in a real situation.
     */

    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;
    }

    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;
```

```
    }

    /**
     * Utility function: inspectCone
     * -----------------------------
     * Called by the manager to simulate the ice-cream-cone inspection
     * process and generate a random bool, where true means the ice cream
     * cone made by the clerk holding his/her attention is good to go, and
     * false means it needs to be remade.
     */

    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;
    }

    /**
     * Thread routines: clerk, cashier, manager, and customer
     * ------------------------------------------------------
     * Each of these four functions provide the script that all of the
     * different players follow.
     */
```

```
    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();
        }
    }

    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;
    }

    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;
    }

    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;
    }

    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;
    }
```

Sample Run Output

```
    myth22> ./ice-cream-store
        Clerk starts to make ice cream cone #0 for customer #2.
    Customer starts to kill time.
        Clerk starts to make ice cream cone #0 for customer #1.
        Clerk starts to make ice cream cone #0 for customer #3.
          Cashier is ready to take customer money.
        Clerk starts to make ice cream cone #0 for customer #0.
    Customer starts to kill time.
```

```
    Customer starts to kill time.
        Clerk starts to make ice cream cone #1 for customer #0.
    Customer starts to kill time.
    Customer starts to kill time.
        Clerk starts to make ice cream cone #1 for customer #2.
        Clerk starts to make ice cream cone #1 for customer #3.
        Clerk starts to make ice cream cone #0 for customer #4.
        Clerk just spent 0.129 seconds making ice cream cone#0 for customer
#0.
      Manager is presented with an ice cream cone.
        Clerk just spent 0.137 seconds making ice cream cone#0 for customer
#1.
        Clerk just spent 0.137 seconds making ice cream cone#1 for customer
#0.
    Customer just killed 0.153 seconds.
    Customer just killed 0.156 seconds.
        Clerk just spent 0.166 seconds making ice cream cone#0 for customer
#3.
        Clerk just spent 0.171 seconds making ice cream cone#0 for customer
#4.
    Customer just killed 0.186 seconds.
    Customer just killed 0.188 seconds.
      Manager spent 0.06 seconds analyzing presented ice cream cone and
APPROVED it.
      Manager is presented with an ice cream cone.
        Clerk just spent 0.197 seconds making ice cream cone#1 for customer
#2.
        Clerk just spent 0.216 seconds making ice cream cone#0 for customer
#2.
    Customer just killed 0.252 seconds.
      Manager spent 0.081 seconds analyzing presented ice cream cone and
APPROVED it.
```

```
    Customer 1 assumes position #0 at the checkout counter.
          Cashier rings up customer 0.
    Customer 1 has checked out and leaves the ice cream store.
      Manager is presented with an ice cream cone.
        Clerk just spent 0.286 seconds making ice cream cone#1 for customer
#3.
      Manager spent 0.023 seconds analyzing presented ice cream cone and
APPROVED it.
    Customer 0 assumes position #1 at the checkout counter.
      Manager is presented with an ice cream cone.
          Cashier rings up customer 1.
    Customer 0 has checked out and leaves the ice cream store.
      Manager spent 0.035 seconds analyzing presented ice cream cone and
APPROVED it.
      Manager is presented with an ice cream cone.
      Manager spent 0.079 seconds analyzing presented ice cream cone and
APPROVED it.
    Customer 4 assumes position #2 at the checkout counter.
          Cashier rings up customer 2.
      Manager is presented with an ice cream cone.
    Customer 4 has checked out and leaves the ice cream store.
      Manager spent 0.051 seconds analyzing presented ice cream cone and
APPROVED it.
      Manager is presented with an ice cream cone.
      Manager spent 0.047 seconds analyzing presented ice cream cone and
APPROVED it.
    Customer 2 assumes position #3 at the checkout counter.
          Cashier rings up customer 3.
    Customer 2 has checked out and leaves the ice cream store.
      Manager is presented with an ice cream cone.
```

```
      Manager spent 0.059 seconds analyzing presented ice cream cone and
APPROVED it.
      Manager inspected a total of 8 ice cream cones before approving a
total of 8.
      Manager leaves the ice cream store.
    Customer 3 assumes position #4 at the checkout counter.
          Cashier rings up customer 4.
    Customer 3 has checked out and leaves the ice cream store.
          Cashier is all done and can go home.
    myth22> ./ice-cream-store
          Cashier is ready to take customer money.
        Clerk starts to make ice cream cone #0 for customer #4.
    Customer starts to kill time.
    Customer starts to kill time.
    Customer starts to kill time.
    Customer starts to kill time.
    Customer starts to kill time.
        Clerk starts to make ice cream cone #0 for customer #1.
        Clerk starts to make ice cream cone #1 for customer #2.
        Clerk starts to make ice cream cone #1 for customer #3.
        Clerk starts to make ice cream cone #1 for customer #1.
        Clerk starts to make ice cream cone #0 for customer #3.
        Clerk starts to make ice cream cone #0 for customer #2.
        Clerk starts to make ice cream cone #0 for customer #0.
        Clerk starts to make ice cream cone #1 for customer #4.
    Customer just killed 0.132 seconds.
        Clerk just spent 0.146 seconds making ice cream cone#1 for customer
#1.
      Manager is presented with an ice cream cone.
```

Clerk just spent 0.146 seconds making ice cream cone#0 for customer #3.

Clerk just spent 0.148 seconds making ice cream cone#0 for customer #4.

Clerk just spent 0.171 seconds making ice cream cone#0 for customer #0.

Clerk just spent 0.186 seconds making ice cream cone#1 for customer #4.

Manager spent 0.056 seconds analyzing presented ice cream cone and REJECTED it.

Clerk starts to make ice cream cone #1 for customer #1.

Manager is presented with an ice cream cone.

Customer just killed 0.208 seconds.

Clerk just spent 0.232 seconds making ice cream cone#0 for customer #1.

Clerk just spent 0.243 seconds making ice cream cone#0 for customer #2.

Clerk just spent 0.247 seconds making ice cream cone#1 for customer #2.

Manager spent 0.049 seconds analyzing presented ice cream cone and REJECTED it.

Clerk starts to make ice cream cone #0 for customer #3.

Manager is presented with an ice cream cone.

Customer just killed 0.26 seconds.

Manager spent 0.022 seconds analyzing presented ice cream cone and APPROVED it.

Manager is presented with an ice cream cone.

Customer just killed 0.276 seconds.

Clerk just spent 0.295 seconds making ice cream cone#1 for customer #3.

Customer just killed 0.297 seconds.

Manager spent 0.095 seconds analyzing presented ice cream cone and APPROVED it.

Customer 0 assumes position #0 at the checkout counter.

Cashier rings up customer 0.

Customer 0 has checked out and leaves the ice cream store.

Manager is presented with an ice cream cone.

Manager spent 0.034 seconds analyzing presented ice cream cone and REJECTED it.

Clerk starts to make ice cream cone #1 for customer #4.

Manager is presented with an ice cream cone.

Clerk just spent 0.193 seconds making ice cream cone#0 for customer #3.

Manager spent 0.08 seconds analyzing presented ice cream cone and APPROVED it.

Manager is presented with an ice cream cone.

Clerk just spent 0.287 seconds making ice cream cone#1 for customer #1.

Manager spent 0.082 seconds analyzing presented ice cream cone and APPROVED it.

Manager is presented with an ice cream cone.

Manager spent 0.031 seconds analyzing presented ice cream cone and APPROVED it.

Customer 2 assumes position #1 at the checkout counter.

Cashier rings up customer 1.

Customer 2 has checked out and leaves the ice cream store.

Manager is presented with an ice cream cone.

Manager spent 0.03 seconds analyzing presented ice cream cone and APPROVED it.

Manager is presented with an ice cream cone.

Manager spent 0.043 seconds analyzing presented ice cream cone and REJECTED it.

Clerk starts to make ice cream cone #0 for customer #3.

Manager is presented with an ice cream cone.

Clerk just spent 0.279 seconds making ice cream cone#1 for customer #4.

Manager spent 0.069 seconds analyzing presented ice cream cone and APPROVED it.

Customer 1 assumes position #2 at the checkout counter.

Cashier rings up customer 2.

Customer 1 has checked out and leaves the ice cream store.

Manager is presented with an ice cream cone.

Manager spent 0.066 seconds analyzing presented ice cream cone and REJECTED it.

Clerk starts to make ice cream cone #1 for customer #4.

Clerk just spent 0.162 seconds making ice cream cone#0 for customer #3.

Manager is presented with an ice cream cone.

Manager spent 0.093 seconds analyzing presented ice cream cone and APPROVED it.

Customer 3 assumes position #3 at the checkout counter.

Cashier rings up customer 3.

Customer 3 has checked out and leaves the ice cream store.

Clerk just spent 0.199 seconds making ice cream cone#1 for customer #4.

Manager is presented with an ice cream cone.

Manager spent 0.022 seconds analyzing presented ice cream cone and REJECTED it.

Clerk starts to make ice cream cone #1 for customer #4.

Clerk just spent 0.085 seconds making ice cream cone#1 for customer #4.

Manager is presented with an ice cream cone.

Manager spent 0.087 seconds analyzing presented ice cream cone and APPROVED it.

```
      Manager inspected a total of 15 ice cream cones before approving a
total of 9.
      Manager leaves the ice cream store.
    Customer 4 assumes position #4 at the checkout counter.
          Cashier rings up customer 4.
          Cashier is all done and can go home.
    Customer 4 has checked out and leaves the ice cream store.
    myth22>
```