CS107 Handout 26
Spring 2008 May 9, 2008

More Concurrency

Written by Julie Zelenski

Semaphore Patterns

One of the latest developments in software engineering is to identify patterns repeated within programs that can be isolated from their context and applied in alternate situations (ideas such as linear search or divide-and-conquer) as a means of cataloguing problem-solving techniques and managing program complexity. It's a pretty handy idea so we at CS107 thought it might be helpful to describe the common semaphore patterns. Hopefully, these descriptions can help you sort out some of the different ways semaphores are used and give you some insight into what options you have for solving different concurrent problems.

Binary lock: The semaphore is used as a single-owner lock. It is initialized to a value of 1 and threads will bracket critical regions with a matched wait and signal on this semaphore. A binary lock semaphore should only ever takes on the value 0 (locked) and 1 (available). This sort of semaphore pattern is used to serialize access to some shared state or resource that only one thread can be using at a time.

Generalized counter: The semaphore is used to count some resource, be it empty buffers, full buffers, clean dishes, or whatever. The semaphore is basically just used as an integer counter, capitalizing on its atomic increment and decrement and the efficient blocking when at zero. The semaphore is initialized to the starting amount of the resource (sometimes 0, sometimes N, it depends). As threads require a resource, they wait on the semaphore to consume one. Other threads (possibly but not necessarily the same threads that consume) signal the semaphore when a new resource available. This sort of semaphore 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) where uncontrolled contention can lead to deadlock.

Binary rendezvous: The semaphore is used to coordinate on some action taking place. Let's say Thread A needs to know that Thread B has finished some task before it can make progress. Rather than having A repeatedly loop and check some global state, a binary rendezvous can be set up between the two. The rendezvous semaphore is initialized to the value 0. When Thread A gets to the point that it needs to know that the task is done, it will 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 signals the semaphore, recording that the task is done, and when A gets to the wait, it will be able to sail right through. 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). The binary rendezvous pattern is often 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 it the paper is refilled) and so on.
If you need a two-way rendezvous where both threads need to wait for the other, you can add another semaphore in the reverse direction, with the wait and signal calls inverted. Be careful that both threads don’t try to wait for the other first and signal afterwards (remember the water example from lecture), else you can quickly land in deadlock!

Generalized rendezvous: A combination of the ideas in the binary rendezvous and the generalized counter. This semaphore is used to record how many times some action has taken place. For example, if Thread A spawned 5 Thread Bs and needs to wait for all of them to finish before going on, a generalized rendezvous can be used. The generalized rendezvous is initialized to the value 0. When Thread A needs to synch up with the others, it will call wait on the semaphore in a loop, one time for each thread it is synching up with. Thread A doesn't care which specific thread of the group has finished, just that another has. If Thread A gets to the rendezvous point before the threads have finished, it will efficiently block, waking to "count" each as it signals, and will only move on when all have checked back in. If all the B threads have finished before A arrives at the rendezvous point, it will be able to quickly decrement the semaphore once for each thread and move on. 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, it will be somewhere between 0 and N at all times. The generalized rendezvous pattern is most often used to regroup after a divided task such as waiting for several network requests to complete or blocking until all pages in a print job have been printed.

Layered construction: Once you have the basic patterns down, you can start to think about how semaphores can be layered into more complex constructions. For example, consider the constrained dining philosopher solution in which a generalized counter is used to constrict the throughput and once inside, binary locks are used for each of the forks. Another layered construct might be a global integer counter with a lock and a binary rendezvous that can do something similar to that of a generalized semaphore. As tasks complete, they could lock and decrement the global counter, and when the counter got to 0, a single signal to the rendezvous could be sent out by the thread that was the last to finish. The combination of a lock with a binary rendezvous could be used to set up a "race": Thread C is waiting for one of Thread A or B to signal. Thread A and B each try to be one who signals the rendezvous. Thread C only expects exactly one signal, thus the lock is used to serialize access so that only the first thread signals, not both. And there are many others you could construct.

The Ice Cream Store Simulation

Before CS107 moves on from the concurrent paradigm, here is one last example that is a bit more complex than the more simple examples we've seen so far. This problem is loosely based on a merged version of two or three old final exam questions.

This program simulates the daily activity in an ice cream store. The actors in this simulation are the clerks who make ice cream cones, the manager who supervises the work of the clerks, the customers who want to buy the cones, and the cashier who rings up each customer. A different thread is launched for each of the players.

Each customer wants to order a few ice cream cones, wait for them to be made, then get in line to pay the cashier, and leave after paying. The customer is in a big hurry and doesn't want to wait for one clerk to make several cones, so instead the customer dispatches one clerk thread independently for each cone. Once the customer has gotten their cones made, they get in line for the cashier and wait for their turn. After paying the cashier, the customer leaves.

Each clerk thread makes exactly one ice cream cone. The clerk scoops up a cone and then has to have the manager take a look at it to make sure it is just right. If the cone doesn't pass muster, it is thrown away and the clerk has to make another. Once a cone passes, the clerk hands the cone to the customer and is done.

The manager sits idle until a clerk needs a cone inspected. When it gets an inspection request, the inspector determines if it passes and lets the clerk know how the cone fared. The manager is done when all cones have been inspected.

In order to avoid fights breaking out, the line for the cashier must be maintained in FIFO order. Thus after getting their cones, a customer "takes a number" to mark their place in the cashier queue. The cashier then always processes customers from this queue in order by number.

The cashier sits idle while there are no customers 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 customers according to position in the queue. Once all customers have paid, the cashier is finished.

We'll walk through solving this problem together in lecture.

```
/**
 * store.c
 * --------
 * An example loosely based on some old final exam questions merged together
 * to show a combination of 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 the work of the clerks, and the cashier who takes the customer's
 * money. There are a variety of 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 tries to the customer in an orderly line, and so on that require
 * use of semaphores to coordinate the activities.
 */

#include <stdio.h>
#include "thread_107.h"

#define NUM_CUSTOMERS 10
#define SECOND 1000000

static void Cashier(void);
static void Clerk(Semaphore done);
static void Manager(int totalNeeded);
static void Customer(int numToBuy);
static void SetupSemaphores(void);
static void FreeSemaphores(void);
static int RandomInteger(int low, int high);
static void MakeCone(void);
static bool InspectCone(void);
static void Checkout(int linePosition);
static void Browse(void);

/* We have many variables accessed by multiple threads in this program and
 * so we have chosen to make them global. In order to keep things tidy, we
 * arrange the globals into coherent structures so that it is easy to
 * understand the relationship between them. This is much preferable to
 * having 12 independent variables declared with no clear indication of
 * their use and grouping.
 */

struct inspection {      // struct of globals for Clerk->Manager rendezvous
    Semaphore available; // lock used to serialize access to the one Manager
    Semaphore requested; // signaled by clerk when cone is ready to inspect
    Semaphore finished;  // signaled by manager after cone has been inspected
    bool passed;         // status of the last inspection
} inspection;

struct line {            // struct of globals for Customer->Cashier line
    Semaphore lock;      // lock used to serialize access to counter
    int nextPlaceInLine; // counter
    Semaphore customers[NUM_CUSTOMERS]; // rendezvous for customer by position
    Semaphore customerReady;// signaled by customer when ready to check out
} line;
```

```
/*
 * The main just sets up all the semaphores and creates all the starting
 * threads. We have one thread per customer, each that is set to buy a
 * random number of cones.  We keep track of the total cones needed so
 * we can pass that information to the manager.
 */

int main(int argc, char **argv)
{
    int i, numCones, totalCones = 0;
    bool verbose = (argc == 2 && (strcmp(argv[1], "-v") == 0));

    InitThreadPackage(verbose);

    SetupSemaphores();
    for (i = 0; i < NUM_CUSTOMERS; i++) {
        char name[32];
        sprintf(name, "Customer %d", i);
        numCones = RandomInteger(1, 4); // each customer wants 1 to 4 cones
        ThreadNew(name, Customer, 1, numCones);
        totalCones += numCones;
    }
    ThreadNew("Cashier", Cashier, 0);
    ThreadNew("Manager", Manager, 1, totalCones);

    RunAllThreads();
    printf("All done!\n");
    FreeSemaphores();
    return 0;
}

/**
 * The manager has a pretty easy job.  He just waits around until a clerk
 * has made an ice cream cone and wants the manager to check it over. The
 * manager efficiently waits for the signal from the clerk and then once
 * awakened, inspects the cone and determines its status, writing to a
 * global variable, and then signals back to the clerk that he has passed
 * judgment. Note that variables like numPerfect and numInspections can and
 * should be local to the Manager since no other thread needs access to them.
 */

static void Manager(int totalNeeded)
{
    int numPerfect = 0, numInspections = 0;

    while (numPerfect < totalNeeded) {
        SemaphoreWait(inspection.requested);
        inspection.passed = InspectCone(); // safe since clerk acquired lock
        numInspections++;
        if (inspection.passed)
            numPerfect++;
        SemaphoreSignal(inspection.finished);
    }
    printf("Inspection success rate %d%%\n", (100*numPerfect)/numInspections);
}
```

```
/*
 * A Clerk thread is dispatched by the customer for each cone they want.
 * The clerk makes the cone and then has to have the manager inspect it.
 * If it doesn't pass, they have to make another. To check with the manager,
 * the clerk has to acquire the exclusive rights to confer with the manager,
 * then signal to wake up the manager, and then wait until they have passed
 * judgment (by writing to a global). We need to be sure that we don't
 * release the inspection lock until we have read the status and are
 * totally done with our inspection.  Once we have a perfect ice cream,
 * we signal back to the originating customer by means of the rendezvous
 * semaphore passed as a parameter to this thread.
 */

static void Clerk(Semaphore done)
{
   bool passed = false;

   while (!passed) {
      MakeCone();
      SemaphoreWait(inspection.available);
      SemaphoreSignal(inspection.requested);
      SemaphoreWait(inspection.finished);
      passed = inspection.passed;
      SemaphoreSignal(inspection.available);
    }

    SemaphoreSignal(done);
}

/*
 * The customer dispatches one thread for each cone desired,
 * then browses around while the clerks make the cones. We create
 * our own local generalized rendezvous semaphore that we pass to
 * the clerks so they can notify us as they finish.  We use
 * that semaphore to "count" the number of clerks who have finished.
 * In the second loop, we wait once for each clerk, which allows
 * us to efficiently block until all clerks check back in.
 * Then we get in line for the cashier by "taking the next number"
 * that is available and signaling our presence to the cashier.
 * When they call our number (signal the semaphore at our place
 * in line), we're done.
 */

static void Customer(int numConesWanted)
{
    int i, myPlace;
    Semaphore clerksDone = SemaphoreNew("Count of clerks done", 0);

    for (i = 0; i < numConesWanted; i++)
        ThreadNew("Clerk", Clerk, 1, clerksDone);

    Browse();

    for (i = 0; i < numConesWanted; i++)
        SemaphoreWait(clerksDone);
    SemaphoreFree(clerksDone);  // this semaphore is not needed anymore

    SemaphoreWait(line.lock);   // binary lock to protect global
    myPlace = line.nextPlaceInLine++; // get number & update line count
    SemaphoreSignal(line.lock);

    SemaphoreSignal(line.customerReady); // signal to cashier we are in line
    SemaphoreWait(line.customers[myPlace]); // wait til checked through
    printf("%s done!\n", ThreadName());
}

/*
 * The cashier just checks the customers through, one at a time,
 * as they become ready.  In order to ensure that the customers get
 * processed in order, we maintain an array of semaphores that allow us
 * to selectively notify a waiting customer, rather than signaling
 * one combined semaphore without any control over which waiter will
 * get notified.
 */

static void Cashier(void)
{
    int i;

    for (i = 0; i < NUM_CUSTOMERS; i++) {
        SemaphoreWait(line.customerReady);
        Checkout(i);
        SemaphoreSignal(line.customers[i]);
    }
}

/*
 * Note carefully the initial values for all the various semaphores.
 * What is the consequence of accidentally starting a lock semaphore
 * out as 0 or 2 instead of 1? What about setting a counting or
 * rendezvous semaphore incorrectly?
 */

static void SetupSemaphores(void)
{
     int i;

     inspection.requested = SemaphoreNew("Inspection Requested", 0);
     inspection.finished = SemaphoreNew("Inspection Finished", 0);
     inspection.available = SemaphoreNew("Manager Available", 1);
     inspection.passed = false;
     line.customerReady = SemaphoreNew("Customer ready", 0);
     line.lock = SemaphoreNew("Line lock", 1);
     line.nextPlaceInLine = 0;
     for (i = 0; i < NUM_CUSTOMERS; i++)
        line.customers[i] = SemaphoreNew("Customer in line", 0);
}
```

```
static void FreeSemaphores(void)
{
     int i;

     SemaphoreFree(inspection.requested);
     SemaphoreFree(inspection.finished);
     SemaphoreFree(inspection.available);
     SemaphoreFree(line.customerReady);
     SemaphoreFree(line.lock);
     for (i = 0; i < NUM_CUSTOMERS; i++)
        SemaphoreFree(line.customers[i]);
}

/* These are just fake functions to stand in for processing steps */

static void MakeCone(void)
{
    ThreadSleep(RandomInteger(0, 3*SECOND));    // sleep random amount
    printf("\t%s making an ice cream cone.\n", ThreadName());
}

static bool InspectCone(void)
{
    bool passed = (RandomInteger(1, 2) == 1);
    printf("\t\t%s examining cone, did it pass? %c\n", ThreadName(),
                           (passed ? 'Y':'N'));
    ThreadSleep(RandomInteger(0, .5*SECOND));    // sleep random amount
    return passed;
}

static void Checkout(int linePosition)
{
   printf("\t\t\t%s checking out customer in line at position #%d.\n",
                    ThreadName(), linePosition);
   ThreadSleep(RandomInteger(0, SECOND));    // sleep random amount
}

static void Browse(void)
{
   ThreadSleep(RandomInteger(0, 5*SECOND));    // sleep random amount
   printf("%s browsing.\n", ThreadName());
}

/*
 * RandomInteger
 * -------------
 * Simple random integer function.
 */

static int RandomInteger(int low, int high)
{
    extern long random();
    long choice;
    int range = high - low + 1;

    PROTECT(choice = random());             // protect non-re-entrant random
    return low + choice % range;
}
```

Output

```
elaine2:/usr/class/cs107/other/thread_examples>store
Customer 0 browsing.
        Clerk making an ice cream cone.
                Manager examining cone, did it pass? N
        Clerk making an ice cream cone.
                Manager examining cone, did it pass? Y
        Clerk making an ice cream cone.
        Clerk making an ice cream cone.
                Manager examining cone, did it pass? N
        Clerk making an ice cream cone.
        Clerk making an ice cream cone.
Customer 1 browsing.
        Clerk making an ice cream cone.
                Manager examining cone, did it pass? Y
Customer 2 browsing.
Customer 3 browsing.
                Manager examining cone, did it pass? N
                Manager examining cone, did it pass? Y
        Clerk making an ice cream cone.
                Manager examining cone, did it pass? N
        Clerk making an ice cream cone.
        Clerk making an ice cream cone.
                Manager examining cone, did it pass? N
        Clerk making an ice cream cone.
        Clerk making an ice cream cone.
                Manager examining cone, did it pass? Y
        Clerk making an ice cream cone.
                Manager examining cone, did it pass? Y
                        Cashier checking out customer in line at position #0.
        Clerk making an ice cream cone.
        Clerk making an ice cream cone.
        Clerk making an ice cream cone.
        Clerk making an ice cream cone.
                Manager examining cone, did it pass? Y
Customer 1 done!
                Manager examining cone, did it pass? Y
                Manager examining cone, did it pass? N
        Clerk making an ice cream cone.
        Clerk making an ice cream cone.
                Manager examining cone, did it pass? N
                Manager examining cone, did it pass? Y
                Manager examining cone, did it pass? Y
                        Cashier checking out customer in line at position #1.
                Manager examining cone, did it pass? Y
                Manager examining cone, did it pass? Y
                Manager examining cone, did it pass? N
        Clerk making an ice cream cone.
                Manager examining cone, did it pass? Y
Customer 2 done!
        Clerk making an ice cream cone.
                Manager examining cone, did it pass? Y
                        Cashier checking out customer in line at position #2.
        Clerk making an ice cream cone.
```

```
                Manager examining cone, did it pass? Y
                        Cashier checking out customer in line at position #3.
Customer 0 done!
                Inspection success rate 61%
Customer 3 done!
All done!
```

Points to Ponder

You can take this starting simulation and extend it with more features and participants to demonstrate even more sophisticated concurrency controls. Here are a few extensions you might want to think through to give yourself practice design more complex coordination:

- What if there were multiple cashiers? What would it take to have the multiple cashiers cooperate on checking customers through? One design might have all cashiers taking customers in order from one combined line. What arrangement will be required between these cashiers so that each customer is checked through exactly once? What if instead there were multiple lines, one for each cashier, and the customer just randomly choose a line and stood in that cashier's line until checked through? In both cases, you need to be careful about the exit conditions so that all cashiers know to exit when all customers have been serviced, whether or not this cashier was the one who actually checked the last one through.

- What if there were multiple managers? A clerk could go to any one of the managers in order to get a cone inspected. Make sure that clerk will flag down any of the available managers rather than waiting for one particular one. Again, like the multiple cashiers, the multiple managers all need to exit when all cones have been inspected, whether or not this manager was the one who actually inspected the last one.

- What if the total number of customers could not be determined at the beginning of the simulation? As it stands, the cashier requires knowing how many customers are coming and it counts the number checked through so it can tell when it is done. What if the customers were randomly generated along the simulation and the number would differ between runs? At some point in the simulation, perhaps the manager will close the store so that no more customers can enter. Now the cashier needs to check through all remaining customers and exit. How could you arrange it so the cashier was able to deal not knowing in advance exactly how many total customers there will be?

- Similarly, what if the total number of cones needed could not be determined at the beginning? The manager uses this info to know when he's done with his inspection duties. Perhaps the customer would browse for a while, then make a decision about how many cones were desired at some later point. How could you arrange it so the manager was able to cope with this uncertainty?