Thread and Semaphore Examples

Handout prose by Julie Zelenski, examples written by Nick Parlante and Julie.

Semaphores

Since all threads run in the same address space, they all have access to the same data and variables. We've seen in lecture that if two threads simultaneously attempt to update a global counter variable, it is possible for their operations to interleave in such way that the global state is not correctly modified. Although such a case may only arise only one time out of thousands, a concurrent program needs to coordinate the activities of multiple threads using something more reliable that just depending on the fact that such interference is rare. The semaphore is designed for just this purpose.

A semaphore is somewhat like an integer variable, but is special in that its operations (increment and decrement) are guaranteed to be atomic—you cannot be halfway through incrementing the semaphore and be interrupted and waylaid by another thread trying to do the same thing. That means you can increment and decrement the semaphore from multiple threads without interference. By convention, when a semaphore is zero it is "locked" or "in use". Otherwise, positive values indicate that the semaphore is available. A semaphore will never have a negative value.

Semaphores are also specifically designed to support an efficient waiting mechanism. If a thread can't proceed until some change occurs, it is undesirable for that thread to be looping and repeatedly checking the state until it changes. In this case semaphore can be used to represent the right of a thread to proceed. A non-zero value means the thread should continue, zero means to hold off. When a thread attempts to decrement a unavailable semaphore (with a zero value), it efficiently waits until another thread increments the semaphore to signal the state change that will allow it to proceed.

Semaphores are usually provided as an ADT by a machine-specific package. As with any ADT, you should only manipulate the variables through the interface routines—in this case `SemaphoreWait` and `SemaphoreSignal` below. There is no single standard thread synchronization facility, but they all look and act pretty similarly.

`SemaphoreWait(Semaphore s)`

If a semaphore value is positive, decrement the value otherwise suspend the thread and **block** on that semaphore until it becomes positive. The thread package keeps track of the threads that are blocked on a particular semaphore. Many packages guarantee FIFO / queue behavior for the unblocking of threads to avoid starvation. Alternately the threads blocked on a semaphore may be stored as a set where the thread manager is free to choose any one. In that case, a thread could theoretically starve, but it's unlikely.

Historically, `P` is a synonym for `SemaphoreWait`. You see, `P` is the first letter in the word *prolagen* which is of course a Dutch word formed from the words *proberen* (to try) and *verlagen* (to decrease).

`SemaphoreSignal(Semaphore s)`

Increment the semaphore value, potentially awakening a suspended thread that is blocked on it. If multiple threads are waiting, it is not deterministic which will be chosen. Also there is no guarantee that any suspended thread will actually begin running immediately when awakened. The awakened thread may just be marked or queued for execution and will run at some later time. `V` historically is a synonym for `SemaphoreSignal` since, of course, *verhogen* means "to increase" in Dutch.

No `GetValue(Semaphore s)` function

One special thing to note about semaphores is that there is no "SemaphoreValue" function in the interface. You cannot look at the value directly, you can only operate on the value through the increment and decrement operations of `Signal` and `Wait`. It isn't really useful to retrieve the value of the semaphore since as you receive the return value there is no guarantee it hasn't been changed in the meantime by another thread.

Semaphore use

In client code, a `SemaphoreWait` call is a sort of checkpoint. If the semaphore is available (i.e. has a positive value) a thread will decrement the value and breeze right through the call to `SemaphoreWait`. If the semaphore is not available, then the thread will efficiently block at the point of the `SemaphoreWait` until the semaphore is available. A call to `SemaphoreWait` is usually balanced by a call to `SemaphoreSignal` to release the semaphore for other threads.

Binary semaphores

A **binary** semaphore can only be 0 or 1. Binary semaphores are most often used to implement a lock that allows only a single thread into a critical section. The semaphore is initially given the value 1 and when a thread approaches the critical region, it waits on the semaphore to decrement the value and "take out" the lock, then signals the semaphore at the end of the critical region to release the lock. Any thread arriving at the critical region while the lock is in use will block when trying to decrement the semaphore, because it is already at 0. When the thread inside the critical region exits, it signals the semaphore and brings its value back up to 1. This allows the waiting thread to now take out the lock and enter the critical section. The result is that at most one thread can enter into the critical section and only after it leaves can another enter. This sort of locking strategy is often used to serialize code that accesses a shared global variable.

The main issues to watch with binary semaphores is ensuring they are initialized to the proper starting state when created and making sure each thread that locks the semaphore is careful to unlock it. You also want to try to keep the critical region as small as possible— only exactly those statements that need to be serialized should be done while holding the lock. If a thread holds the lock during a lot of other operations that aren't accessing any shared data, it is unnecessary holding up all the other threads that need to acquire that lock.

General semaphores

A **general** semaphore can take on any non-negative value. General semaphores are used for "counting" tasks such as creating a critical region that allows a specified number of threads to enter. For example, if you want at most four threads to be able to enter a section, you could protect it with a semaphore and initialize that semaphore to four. The first four threads will be able to decrement the semaphore and enter the region, but at that point, the semaphore will be zero and any other threads will block outside the critical region until one of the current threads leaves and signals the semaphore.

You can also use a general semaphore for representing the quantity of an available resource. Let's say you need to limit the number of simultaneously open file descriptors among multiple threads. You could initialize a general semaphore to the maximum number of open file descriptors and each thread that wants to open a file needs to wait on the semaphore first. If the max hasn't yet been reached, the semaphore will have a positive value and the thread will be able to breeze right through the wait, decrement the semaphore and thus open a file. If the max has been reached, the semaphore value will be zero and thread will block until another thread closes a file, releasing a resource, and incrementing the semaphore that allows others to proceed.

A general semaphore can also be used to implement a form of rendezvous between threads, such as when Thread2 needs to know that Thread1 is done with something before proceeding. A rendezvous semaphore is usually initialized to zero. Thread1 waits on that semaphore (and thus immediately blocks since the value starts at zero) until Thread2 signals the semaphore when ready. If you need to rendezvous among several threads, you could have Thread1 wait several times, once for each of the threads that will signal when ready. In this case, the semaphore is "counting" the number of times an action occurred.

Global variables

You will find that concurrent programs tend to use global variables, something you may have been trained to believe is evil in its purest form and thus will balk a bit at this kind of design. It is characteristic of multi-threaded programs that data is made globally visible to allow multiple threads to access it. This is appropriate for the data that is shared and worked upon by more than one thread. Global variables tend to be the easiest way to share data in a concurrent program. However, you inherit all the usual

downsides of globals—keeping track of who changes the data where is difficult, it leads to routines that have lots of interdependencies other than what is indicated by the parameter lists, and so on.

As an alternative, you can declare variables as local variables within a function (most usually the main function) and then pass pointers to those variables as arguments to the new threads. This avoids the global variables and all their attendant risks and gives you direct control and documentation about which routines have access to these pieces of data. The downside is longer argument lists for the functions and more complicated variable management. You also need to be very careful here—if you are going to pass a pointer to a stack variable from one thread's stack to an another, you need to be absolutely positive that the original stack frame will remain valid for the entire time the other thread is using the pointers it was given. This can be tricky! Since the main function exists for the lifetime of the entire program, its local variables aren't at risk, but be very wary when trying to do this with any other function's local variables.

We don't prefer one approach to the exclusion of the other and so our examples will show a mix of styles and you are free to adopt the one that works best for you.

Binary Semaphore Example

The canonical use of a semaphore is a lock associated with some resource so that only one thread at a time has access to the resource. In the example below, we have one piece of global data, the number of tickets remaining to sell, that we want to coordinate the access by multiple threads. In this case, a binary semaphore serves as a lock to guarantee that at most one thread is examining or changing the value of the variable at any given time.

When the program is run, it creates a certain number of threads that attempt to sell all the available tickets. This code is written with the thread package we will be using on the Sun workstations.

```
/*
 * ticketSeller.c
 * --------------
 * A very simple example of a critical section that is protected by a
 * semaphore lock.  There is a global variable numTickets which tracks the
 * number of tickets remaining to sell.  We will create many threads that all
 * will attempt to sell tickets until they are all gone. However, we must
 * control access to this global variable lest we sell more tickets than
 * really exist.  We have a semaphore lock that will only allow one seller
 * thread to access the numTickets variable at a time.  Before attempting to
 * sell a ticket, the thread must acquire the lock by waiting on the semaphore
 * and then release the lock when through by signalling the semaphore.
 */

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

```
#define     NUM_TICKETS       35
#define     NUM_SELLERS       4
```

```
/*
 * The ticket counter and its associated lock will be accessed
 * all threads, so made global for easy access.
 */

static int numTickets = NUM_TICKETS;
static Semaphore ticketsLock;

/*
 * Our main is creates the initial semaphore lock in an unlocked state
 * (one thread can immediately acquire it) and sets up all of
 * the ticket seller threads, and lets them run to completion.  They
 * should all finish when all tickets have been sold. By running with the
 * -v flag, it will include the trace output from the thread library.
 */
void main(int argc, char **argv)
{
    int i;
    char name[32];
    bool verbose = (argc == 2 && (strcmp(argv[1], "-v") == 0));

    InitThreadPackage(verbose);

    ticketsLock = SemaphoreNew("Tickets Lock", 1);
    for (i = 0; i < NUM_SELLERS; i++) {
      sprintf(name, "Seller #%d", i);     // give each thread a distinct name
      ThreadNew(name, SellTickets, 0);
    }
    RunAllThreads();                      // Let all threads loose
    SemaphoreFree(ticketsLock);           // to be tidy, clean up
    printf("All done!\n");
}
```

```
/*
 * SellTickets
 * -----------
 * This is the routine forked by each of the ticket-selling threads.
 * It will loop selling tickets until there are no more tickets left
 * to sell.  Before accessing the global numTickets variable,
 * it acquires the ticketsLock to ensure that our threads don't step
 * on one another and oversell on the number of tickets.
 */

static void SellTickets(void)
{
   bool done = false;
   int numSoldByThisThread = 0;     // local vars are unique to each thread

   while (!done) {

      /* imagine some code here which does something independent of
       * the other threads such as working with a customer to determine
       * which tickets they want.  Simulate with a small random delay
       * to get random variations in output patterns.
       */
      RandomDelay(0,2);

      SemaphoreWait(ticketsLock);   // ENTER CRITICAL SECTION
      if (numTickets == 0) {        // here is safe to access numTickets
         done = true;   // a "break" here instead of done variable
                        // would be an error- why?
      } else {
         numTickets--;
         numSoldByThisThread++;
         printf("%s sold one (%d left)\n", ThreadName(), numTickets);
      }
      SemaphoreSignal(ticketsLock); // LEAVE CRITICAL SECTION
   }

    printf("%s noticed all tickets sold! (I sold %d myself) \n",
         ThreadName(), numSoldByThisThread);
}


/*
 * RandomDelay
 * -------------
 * This is used to put the current thread to sleep for a little
 * bit to simulate some activity or perhaps just to vary the
 * execution patterns of the thread scheduling.  The low and high
 * limits are expressed in microseconds, the thread will sleep
 * at least the lower limit, and perhaps as high as upper limit
 * (or even more depending on the contention for the processors).
 */
static void RandomDelay(int atLeastMicrosecs, int atMostMicrosecs)
{
    long choice;
    int range = atMostMicrosecs - atLeastMicrosecs;

    PROTECT(choice = random());                     // protect non-re-entrancy
    ThreadSleep(atLeastMicrosecs + choice % range); // put thread to sleep
}
```

Output

```
epic18:/usr/class/cs107/other/thread_examples>ticketSeller
Seller #1 sold one (34 left)
Seller #0 sold one (33 left)
Seller #1 sold one (32 left)
Seller #1 sold one (31 left)
Seller #1 sold one (30 left)
Seller #1 sold one (29 left)
Seller #1 sold one (28 left)
Seller #2 sold one (27 left)
Seller #3 sold one (26 left)
Seller #2 sold one (25 left)
Seller #3 sold one (24 left)
Seller #2 sold one (23 left)
Seller #0 sold one (22 left)
Seller #1 sold one (21 left)
Seller #2 sold one (20 left)
Seller #0 sold one (19 left)
Seller #1 sold one (18 left)
Seller #1 sold one (17 left)
Seller #3 sold one (16 left)
Seller #2 sold one (15 left)
Seller #0 sold one (14 left)
Seller #0 sold one (13 left)
Seller #1 sold one (12 left)
Seller #3 sold one (11 left)
Seller #2 sold one (10 left)
Seller #0 sold one (9 left)
Seller #0 sold one (8 left)
Seller #1 sold one (7 left)
Seller #3 sold one (6 left)
Seller #2 sold one (5 left)
Seller #0 sold one (4 left)
Seller #1 sold one (3 left)
Seller #1 sold one (2 left)
Seller #1 sold one (1 left)
Seller #1 sold one (0 left)
Seller #3 noticed all tickets sold! (I sold 5 myself)
Seller #2 noticed all tickets sold! (I sold 7 myself)
Seller #1 noticed all tickets sold! (I sold 15 myself)
Seller #0 noticed all tickets sold! (I sold 8 myself)
All done!
```

Note that each time you run it, different output will result because the threads will not be scheduled in exactly the same way. Maybe next time `Seller 0` will sell most of the tickets or `Seller 3` will finish last. But it should always be true that exactly `35` tickets are sold, no more, no less, and that's what our use of the semaphore lock is designed to ensure.

Reader-Writer example

In this classic Reader-Writer problem, there are two threads exchanging information through a fixed size buffer. The Writer thread fills the buffer with data whenever there's room for more. The Reader thread reads data from the buffer and prints it. Both threads have a situation where they should block. The writer blocks when the buffer is full and the reader blocks when the buffer is empty. The problem is to get them to cooperate nicely and block efficiently when necessary.

For this problem, we will use "generalized semaphores" where the value can be any non-negative number. Zero still means "locked" and any other value means "available". The code cannot look at the value of a generalized semaphore explicitly, you can only call `SemaphoreWait` and `SemaphoreSignal` which in turn depend on the value.

There is a shared, fixed-size buffer. The reader reads starting at `readPt` and the writer writes at `writePt`. No locks are required to protect these integers because only one thread concerns itself with either. The semaphores ensure that the writer only writes at `writePt` when there is space available and similarly for the reader and `readPt`. This program is written using no global variables, but instead declaring the variables in main and passing their address to the new threads.

```
/*
 * readerWriter.c
 * --------------
 * The canonical consumer-producer example.  This version has just one reader
 * and just one writer (although it could be generalized to multiple readers/
 * writers) communicating information through a shared buffer.  There are two
 * generalized semaphores used, one to track the num of empty buffers, another
 * to track full buffers. Each is used to count, as well as control access.
 */

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

#define NUM_TOTAL_BUFFERS 5
#define DATA_LENGTH 20

/*
 * Initially, all buffers are empty, so our empty buffer semaphore starts
 * with a count equal to the total number of buffers, while our full buffer
 * semaphore begins at zero.  We create two threads:  one to read and one
 * to write, and then start them off running.  They will finish after all
 * data has been written & read. By running with the -v flag, it will include
 * the trace output from the thread library.
 */

void main(int argc, char **argv)
{
    bool verbose = (argc == 2 && (strcmp(argv[1], "-v") == 0));
    Semaphore emptyBuffers, fullBuffers;   // semaphores used as counters
    char buffers[NUM_TOTAL_BUFFERS];       // the shared buffer

    InitThreadPackage(verbose);

    emptyBuffers = SemaphoreNew("Empty Buffers", NUM_TOTAL_BUFFERS);
    fullBuffers = SemaphoreNew("Full Buffers", 0);

    ThreadNew(“Writer”, Writer, 3, buffers, emptyBuffers, fullBuffers);
    ThreadNew(“Reader”, Reader, 3, buffers, emptyBuffers, fullBuffers);
    RunAllThreads();
    SemaphoreFree(emptyBuffers);
    SemaphoreFree(fullBuffers);
    printf("All done!\n");
}
```

```
/*
 * Writer
 * ------
 * This is the routine forked by the Writer thread.  It will loop until
 * all data is written.  It prepares the data to be written, then waits
 * for an empty buffer to be available to write the data to, after which
 * it signals that a full buffer is ready.
 */

static void Writer(char buffers[],
                   Semaphore emptyBuffers,
                   Semaphore fullBuffers)
{
   int i, writePt = 0;
   char data;

   for (i = 0; i < DATA_LENGTH; i++) {
      data = PrepareData();           // go off & get data ready
      SemaphoreWait(emptyBuffers);    // now wait til an empty buffer avail
      buffers[writePt] = data;        // put data into buffer
      printf("%s: buffer[%d] = %c\n", ThreadName(), writePt, data);
      writePt = (writePt + 1) % NUM_TOTAL_BUFFERS;
      SemaphoreSignal(fullBuffers);   // announce full buffer ready
   }
}


/*
 * Reader
 * ------
 * This is the routine forked by the Reader thread.  It will loop until
 * all data is read.  It waits until a full buffer is available and the
 * reads from it, signals that now an empty buffer is ready, and then
 * goes off and processes the data.
 */
static void Reader(char buffers[],
                   Semaphore emptyBuffers,
                   Semaphore fullBuffers)
{
   int i, readPt = 0;
   char data;

   for (i = 0; i < DATA_LENGTH; i++) {
      SemaphoreWait(fullBuffers);     // wait til something to read
      data = buffers[readPt];         // pull value out of buffer
      printf("\t\t%s: buffer[%d] = %c\n", ThreadName(), readPt, data);
      readPt = (readPt + 1) % NUM_TOTAL_BUFFERS;
      SemaphoreSignal(emptyBuffers);  // announce empty buffer
      ProcessData(data);              // now go off & process data
   }
}
```

```
/*
 * ProcessData
 * -----------
 * This just stands in for some lengthy processing step that might be
 * required to handle the incoming data. Processing the data can be done by
 * many reader threads simultaneously since it doesn't access any global state
 */

static void ProcessData(char data)
{
    ThreadSleep(RandomInteger(0, 500));   // sleep random amount
}

/*
 * PrepareData
 * -----------
 * This just stands in for some lengthy processing step that might be
 * required to create the data.  Preparing the data can be done by many writer
 * threads simultaneously since it doesn't access any global state.  The data
 * value is just randomly generated in our simulation.
 */

static char PrepareData(void)
{
   ThreadSleep(RandomInteger(0, 500));    // sleep random amount
   return RandomInteger('A', 'Z');        // return random character
}
```

Output

```
   saga21:/usr/class/cs107/other/thread_examples>readerWriter
   Writer: buffer[0] = M
                           Reader: buffer[0] = M
   Writer: buffer[1] = C
                           Reader: buffer[1] = C
   Writer: buffer[2] = C
                           Reader: buffer[2] = C
   Writer: buffer[3] = D
                           Reader: buffer[3] = D
   Writer: buffer[4] = T
                           Reader: buffer[4] = T
   Writer: buffer[0] = X
   Writer: buffer[1] = U
                           Reader: buffer[0] = X
                           Reader: buffer[1] = U
   Writer: buffer[2] = D
   Writer: buffer[3] = Y
   Writer: buffer[4] = C
                           Reader: buffer[2] = D
   Writer: buffer[0] = Q
                           Reader: buffer[3] = Y
   Writer: buffer[1] = I
                           Reader: buffer[4] = C
                           Reader: buffer[0] = Q
   Writer: buffer[2] = D
   Writer: buffer[3] = L
                           Reader: buffer[1] = I
   Writer: buffer[4] = W
                           Reader: buffer[2] = D
   Writer: buffer[0] = D
```

```
                        Reader: buffer[3] = L
Writer: buffer[1] = M
                        Reader: buffer[4] = W
Writer: buffer[2] = Y
                        Reader: buffer[0] = D
Writer: buffer[3] = P
                        Reader: buffer[1] = M
                        Reader: buffer[2] = Y
                        Reader: buffer[3] = P
Writer: buffer[4] = R
                        Reader: buffer[4] = R
All done!
```

Another run won't necessary get the same output, but the reader should always list the chars in the same order as the writer and there should never be more than 5 writes before a read and vice versa (since we only have 5 buffers to use).

Making water

Another simple concurrency problem is simulating the process of hooking up hydrogen atoms with oxygen atoms to form water molecules. Each atom is represented by a separate thread, we will need to associate two hydrogen threads with one oxygen thread to make a water molecule, then all three threads exit together.

We will use two general semaphores, one to count the number of hydrogen threads and one for the number of oxygen threads. A hydrogen thread will wait to consume one of the oxygen and then signal to raise the count of hydrogen to communicate to the oxygen thread. An oxygen thread will wait for two hydrogen to come ready and then signal the oxygen count twice to let them know oxygen is ready. This is an example of a "rendezvous"—we are signalling a general semaphore to record the action of one thread and another thread can wait on it to meet up with it.

```
/*
 * water.c
 * -------
 * A simple deadlock example.  This version will quickly get stuck when all
 * the oxygen wait for hydrogen to get ready at the same time the hydrogen
 * is waiting for the oxygen to get ready. A simple change in order of
 * signal would solve this problem.
 */

#include    "thread_107.h"
#define     NUM_WATER         10

void main(int argc, char **argv)
{
   int i;
   bool verbose = (argc == 2 && (strcmp(argv[1], "-v") == 0));
   Semaphore oxygenReady, hydrogenReady;   // semaphores used as counters

   InitThreadPackage(verbose);
   oxygenReady = SemaphoreNew("Oxygen Ready", 0);
   hydrogenReady = SemaphoreNew("Hydrogen Ready", 0);

   for (i = 0; i < NUM_WATER; i++)
      ThreadNew(“Oxygen”, Oxygen, 2, oxygenReady, hydrogenReady);
   for (i = 0; i < 2 * NUM_WATER; i++)
      ThreadNew(“Hydrogen”, Hydrogen, 2, oxygenReady, hydrogenReady);

   RunAllThreads();

   printf("All done!\n");
   SemaphoreFree(oxygenReady);
   SemaphoreFree(hydrogenReady);
}
```

```
static void Hydrogen(Semaphore oxygenReady, Semaphore hydrogenReady)
{
   SemaphoreWait(oxygenReady);
   SemaphoreSignal(hydrogenReady);
}

static void Oxygen(Semaphore oxygenReady, Semaphore hydrogenReady)
{
   SemaphoreWait(hydrogenReady);
   SemaphoreWait(hydrogenReady);
   SemaphoreSignal(oxygenReady);
   SemaphoreSignal(oxygenReady);
   printf("Water made!\n");
}
```

Deadlock

In the above program, things will quickly grind to a halt as all threads start up and immediately start waiting on one of the two counters, which all start as zero. Every thread is then waiting for an action to be taken by another thread— we call this situation "deadlock." In the water program, we can eliminate the problem by changing the order for one of the elements. Somebody has to make the first move. If hydrogen first signals hydrogen available and then waits on the oxygen counter, it will break the deadlock because raising the hydrogen count allows the oxygen thread to get through acquiring hydrogen to signal the oxygen needed by the hydrogen thread, which will allow both the elements to move on.

In a simple case like this where the deadlock is immediately obvious and infinitely reproducible, it is easy to detect and fix. However there are many more subtle cases where the deadlock may happen only rarely and can be difficult to track down and remove.

The water program is a good one to practice a little concurrent debugging on. Try running it under `gdb` and when it gets stuck, interrupt the program and use the debugging routines to print all threads and semaphores to get a picture of what is happening during execution.

Dining Philosophers

Another classic concurrency problem concerns a group of philosophers seated about a round table eating spaghetti. There are the same total number of forks as there are philosophers and one fork is placed between every two philosophers—that puts a single fork to the left and right of each philosopher. The philosophers run the traditional `think-eat` loop. After he sits quietly thinking for a while, a philosopher gets hungry. To eat, a philosopher grabs the fork to the left, grabs the fork to the right, eats for a time using both forks, and then replaces the forks and goes back to thinking. (There's a related problem, the Dining Programmers, where you have group of programmers sitting around a round table eating sushi with one chopstick between every two programmers.)

There is a possible deadlock condition if the philosophers are allowed to grab forks freely. The deadlock occurs if no one is eating, and then all the philosophers grab the fork to their left, and then look over and wait for the right fork.

Solution

One simple and safe solution is to restrict the number of philosophers allowed to even try to eat at once. If you only allow (n-1) of the philosophers to try to eat, then you can show that the deadlock situation cannot occur. This correctness comes at the cost of allowing slightly less spaghetti throughput than without the restriction. The restrict-competition-to-avoid-the-deadlock can be a staple technique for avoiding deadlock. What is another way to avoid the deadlock? Hint: don't have all the philosophers follow the exact same program.

```
/*
 * dining.c
 * --------
 * The classic Dining Philosophers example.  Allow a group of dining
 * philosophers to eat at a round sharing the forks between each person.
 * Each person needs both forks to eat.
 */

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

#define NUM_DINERS 5
#define EAT_TIMES 3

/* Macros to conveniently refer to forks to left and right of each person */

#define LEFT(philNum) (philNum)
#define RIGHT(philNum) (((philNum)+1) % NUM_DINERS)

/*
 * Our main is creates a semaphore for every fork in an unlocked state
 * (one philosopher can immediately acquire each fork) and sets up the
 * numEating semaphore to only allow N-1 philosophers to try and grab
 * their forks.  Each philosopher runs its own thread.  They should
 * finish after getting their fill of spaghetti. By running with the
 * -v flag, it will include the trace output from the thread library.
 */

void main(int argc, char **argv)
{
   int i;
   char name[32];
   bool verbose = (argc == 2 && (strcmp(argv[1], "-v") == 0));
   Semaphore fork[NUM_DINERS];   // semaphore to control access per fork
   Semaphore numEating;          // to restrict contention for forks

   InitThreadPackage(verbose);

   for (i = 0; i < NUM_DINERS; i++) {  // Create all fork semaphores
      sprintf(name, "Fork %d", i);
```

```
   fork[i] = SemaphoreNew(name, 1); // all forks start available
}
numEating = SemaphoreNew(“Num Eating”, NUM_DINERS - 1);
```

```
   for (i = 0; i < NUM_DINERS; i++) {  // Create all philosopher threads
      sprintf(name, "Philosopher %d", i);
      ThreadNew(name, Philosopher, 3, numEating, fork[LEFT(i)], fork[RIGHT(i)]);
   }

   RunAllThreads();
   printf("All done!\n");

   SemaphoreFree(numEating);
   for (i = 0; i < NUM_DINERS; i++) SemaphoreFree(fork[i]);
}

/*
 * Philosopher
 * -----------
 * This is the routine run in each of the philosopher threads.  Each runs in
 * ean at-think loop where pondering for a while builds up a big hunger.
 */
static void Philosopher(Semaphore numEating, Semaphore leftFork,
                        Semaphore rightFork)
{
   int i;

   for (i = 0; i < EAT_TIMES; i++) {
      Think();
      Eat(numEating, leftFork, rightFork);
   }
}


static void Think(void)
{
    printf("%s thinking!\n", ThreadName());
    RandomDelay(10000,50000);             // "think" for random time
}


/*
 * We first wait on the availability of an opportunity to eat, and
 * only then do we attempt to grab our left & right forks and chow
 * some spaghetti.  Notice that we let go of the locks in the reverse
 * order that we acquire them, so that we don't signal another
 * philosopher to eat until after we have put down both our forks.
 */
static void Eat(Semaphore numEating, Semaphore leftFork, Semaphore rightFork)
{
    SemaphoreWait(numEating);    // wait until can try to get forks
    SemaphoreWait(leftFork);           // get left
    SemaphoreWait(rightFork);          // get right

    printf("%s eating!\n", ThreadName());
    RandomDelay(10000,50000);             // "eat" for random time

    SemaphoreSignal(leftFork);         // let go
    SemaphoreSignal(rightFork);
    SemaphoreSignal(numEating);
}
```

Output

```
elaine24:/usr/class/cs107/other/thread_examples> dining
Philosopher 3 thinking!
Philosopher 4 thinking!
Philosopher 0 thinking!
Philosopher 1 thinking!
Philosopher 2 thinking!
Philosopher 2 eating!
Philosopher 0 eating!
Philosopher 0 thinking!
Philosopher 4 eating!
Philosopher 2 thinking!
Philosopher 4 thinking!
Philosopher 0 eating!
Philosopher 2 eating!
Philosopher 2 thinking!
Philosopher 0 thinking!
Philosopher 3 eating!
Philosopher 0 eating!
Philosopher 3 thinking!
Philosopher 2 eating!
Philosopher 4 eating!
Philosopher 4 thinking!
Philosopher 1 eating!
Philosopher 3 eating!
Philosopher 3 thinking!
Philosopher 1 thinking!
Philosopher 4 eating!
Philosopher 1 eating!
Philosopher 3 eating!
Philosopher 1 thinking!
Philosopher 1 eating!
All done!
```