The Cook, the Dishwasher, and the Customers

Example 1:

The problem is to simulate a buffet cafeteria's operation. In particular, we want to simulate how the number of dishes in the cafeteria affects the overall operation. The actors in our world are a single cook, a group of customers, and a single dishwasher. Conditions are that at the beginning of each day, all the dishes are clean, and all of the dishes must be cleaned by the end of each day.

Acting Roles

Cook	For every `TOTAL_DISHES_TO_MAKE` dish that will be consumed, the cook gets a clean dish, puts some food on it, and leaves it on the buffet table for a customer
Customer	Each of the `NUM_CUSTOMERS` customers makes `TRIPS_PER_CUSTOMER` trips to the buffet table. A complete trip consists of getting a full dish from the table, eating the meal, and placing the dirty dish on the dirty dish rack. When customers get full they leave the cafeteria.
Dishwasher	The dishwasher's basic actions consist of waiting at the dirty dish rack for dishes and then cleaning the dishes as they come in. The simple version of this problem has the dishwasher washing the dishes by hand so that only one dish can be cleaned at a time. We will look at a more complicated version afterwards.

The simple simulation

Each acting role in our cafeteria can be represented by a separate function and each actor is represented by a single thread. The functions themselves are basically direct translations from the prose above into code form.

We will use three general semaphores to "count" our resources: `cleanDishes`, `fullDishes`, and `dirtyDishes`. We start out our dirty and full counts as 0 and our clean count at all the dishes. As dishes pass from cook to customer to dishwasher the various semaphores will be incremented and decremented to track the states of our resources.

```
/*
 * simplecafe.c
 * Author: Julie Zelenski, April 96
 * ------------
 * The simplified cafeteria example.  In this example, we have one
 * cook, one dishwasher, and some number of customers who will each make
 * the same number of trips to the buffet table.  We've got a fixed number
 * of dishes to use.  When a dish is clean, the cook can serve up another
 * plate for a hungry customer, who eats it, and then puts the dish on
 * the dirty dish rack for the dishwasher to clean.  In this version, the
 * dishwasher immediately washes each disk one by one, without trying to
 * consolidate loads.
 */

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

#define NUM_CUSTOMERS 4
#define TRIPS_PER_CUSTOMER 2
#define NUM_DISHES 4

/* Semaphores are global in scope since all threads access them. */
static Semaphore fullDishes, cleanDishes, dirtyDishes;

/*
 * Cook
 * ----
 * This is the routine forked by the Cook thread.  The cook's job will
 * be to serve up all the dishes.  It will wait until there is a clean
 * dish available, then pile some food on it and announce a full dish
 * is ready.  The cook is very similar to the writer thread in the
 * Reader/Writer example.  The cook finishes when all customers have
 * been completely served.
 */
void Cook(void)
{
   int i, numDishesToMake;

   /* We can directly calculate the number of dishes to make */
   numDishesToMake = NUM_CUSTOMERS * TRIPS_PER_CUSTOMER;

   for (i=0; i< numDishesToMake; i++) {
      SemaphoreWait(cleanDishes);   // Wait till someone holds out a tray
      // Put some slop onto the tray
      printf("Cook served one, brings total served to %d\n", i+1);
      SemaphoreSignal(fullDishes);  // Ring the bell to start drooling
   }

   printf("COOK THREAD FINISHED\n");
}
```

```
/*
 * Customer
 * --------
 * This is the routine forked by all the Customer threads.  The customer's
 * keeps coming up to the buffer for numTrips iteration to get a plate
 * of yummy food, which the customer quickly eats and then places the
 * dirty dish on the busyboy's cart for the dishwasher to wash.  Each
 * customer asks for the same number of dishes and exits when it got
 * them all.
 */

void Customer(int custNum)
{
   int i;

   for (i=0; i< TRIPS_PER_CUSTOMER; i++) {
      SemaphoreWait(fullDishes);    // wait till cook serves something
      printf("Customer #%d served on trip %d\n", custNum, i+1);
      SemaphoreSignal(dirtyDishes); // announce a dirty dish is ready
   }

   printf("CUSTOMER #%d THREAD FINISHED \n", custNum);
}

/*
 * Dishwasher
 * ----------
 * This is the routine forked by the Dishwasher thread.  The dishwasher's
 * job will to be washed the used dishes. It will wait until there is
 * a dirty dish available, wash it, and then announce a clean dish. It's
 * position is fairly similar to that of the Reader in the Reader/Writer
 * example.  When all dishes have been washed, the dishwasher's job is
 * done.
 */

void Dishwasher(void)
{
    int i, numDishesToClean;

    numDishesToClean = NUM_CUSTOMERS * TRIPS_PER_CUSTOMER;

    for (i=0; i< numDishesToClean; i++) {
      SemaphoreWait(dirtyDishes);   // wait till something dirty
      printf("Dishwasher washed one, brings total washed to %d\n", i+1);
      SemaphoreSignal(cleanDishes);
    }

    printf("DISHWASHER THREAD FINISHED\n");
}
```

```
/*
 * Our main creates a group of customer threads and the single cook
 * and dishwasher threads. The cleanDishes semaphore starts out with
 * all dishes being clean, and the number of full and dirty dishes
 * starts out at zero.
 */

int main(int argc, char **argv)
{
   int i;
   char name[32];
   bool verbose = (argc == 2 && (strcmp(argv[1], "-v") == 0));

   InitThreadPackage(verbose);

   for (i = 0; i < NUM_CUSTOMERS; i++) {
      sprintf(name, "Customer %d", i);
      ThreadNew(name, Customer, 1, i);
   }
   ThreadNew("Cook", Cook, 0);
   ThreadNew("Dishwasher", Dishwasher, 0);

   cleanDishes = SemaphoreNew("Clean Dishes", NUM_DISHES);
   fullDishes = SemaphoreNew("Full Dishes", 0);
   dirtyDishes = SemaphoreNew("Dirty Dishes", 0);

   RunAllThreads();
   SemaphoreFree(cleanDishes);
   SemaphoreFree(fullDishes);
   SemaphoreFree(dirtyDishes);

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

Simulation Take 2

To make the dishwasher's job a bit less tedious a Hobart will be purchased. The Hobart adds the following to the picture:

1. The Hobart can hold at most `MAX_LOAD` dishes per load, but, in an effort to conserve water, it should not be run unless there are at least `MIN_LOAD` dirty dishes.

2. At the end of the day, however, all of the remaining dirty dishes must be washed, even if this number is below the `MIN_LOAD` limit. If there happens to be more than `MAX_LOAD` dirty dishes then the dishwasher will run as many loads as needed to get all the dishes washed.

We consider each of the new constraints in turn:

1. An approach to taking into account the first constraint is to modify the dishwasher so that the maximum possible number of dishes is cleaned each time through the loop. How do we know how many dishes are ready to be cleaned? The `dirtyDishes` semaphore tracks the number of dirty dishes, but we cannot directly access its value. Instead, we can add an integer variable `numDirtyDishes` that can be directly read.

Since we expect this variable to be modified by both customers and the dishwasher, we will need to protect access to it with a locking semaphore. We will also add a `startLoad` semaphore that a customer can use to signal the dishwasher to indicate that the minimum number of dishes for a load are in the rack and that it's time to run the Hobart.

2. The second constraint can be added with the addition of another variable tracking the number of customers still in the cafeteria. This variable is initially equal to the number of customers and is decremented by each customer as they leave. When the last customer leaves, the last load of the night needs to be run (if it has not already been scheduled to run).

Note that without the second constraint, we would introduce the possibility of deadlock. Consider this output:

```
TRIPS_PER_CUSTOMER = 1, NUM_CUSTOMERS = 4, NUM_DISHES = 4
MIN_LOAD = 2, MAX_LOAD = 3

Customer 0 served on trip 1
CUSTOMER 0 THREAD FINISHED
Customer 1 served on trip 1
CUSTOMER 1 THREAD FINISHED
Customer 2 served on trip 1
CUSTOMER 2 THREAD FINISHED
Dishwasher washed 3, brings total washed to 3
Customer 3 served on trip 1
CUSTOMER 3 THREAD FINISHED
```

Deadlock occurs because the dishwasher does not know that the last customer has finished. The dishwasher now waits for the minimum number of dishes to appear, which will never happen since there are no more customers.

The proposed working solution is then to add two new global variables tracking the number of active customers and the number of dirty dishes, `numDirtyDishes` and `numActiveCustomers`.

Since these two variables are accessed by multiple threads, each has a binary semaphore associated with it to control access to it. We remove the `dirtyDishes` semaphore and instead have a `startLoad` semaphore that will be used to inform the Dishwasher its time to run a load. `startLoad` will be signaled when a customer puts a dish into the rack that pushes the current dirty dish count past `MIN_LOAD`. It is also possibly signaled by the last customer to leave the cafeteria for the remaining odd load.

General Solution

```
/*
 * cafe.c
 * -------
 * The more complex cafeteria example.  In this example, we have one
 * cook, one dishwasher, and some number of customers who will each make
 * the same number of trips to the buffet table.  We've got a fixed number
 * of dishes to use.  When a dish is clean, the cook can serve up another
 * plate for a hungry customer, who eats it, and then puts the dish on
 * the dirty dish rack for the dishwasher to clean.  In this version, the
 * dishwasher can wash multiple dishes, up to MAX_LOAD, at a time.  Also
 * to conserve water, the dishwasher won't run a load unless there are
 * MIN_LOAD dirty dishes.  The only exception is that at the end of the day,
 * we wash whatever dishes are left, even if it is below the min load.
 */

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

#define NUM_CUSTOMERS 25
#define TRIPS_PER_CUSTOMER 4
#define NUM_DISHES 15
#define MIN_LOAD 2
#define MAX_LOAD 5
#define MIN(x, y)  ((x) <= (y) ? (x) : (y))

static Semaphore fullDishes, cleanDishes;
static Semaphore startLoad, dirtyLock, activeLock;

static int numDirtyDishes, numActiveCustomers;


/*
 * Cook
 * ----
 * This is the routine forked by the Cook thread.  The cook's job will
 * be to serve up all the dishes.  It will wait until there is a clean
 * dish available, then pile some food on it and announce a full dish
 * is ready.  The cook is very similar to the writer thread in the
 * Reader/Writer example.  The cook finishes when all customers have
 * been completely served.
 */

void Cook(void)
{
   int i, numDishesToMake;

   /* We can directly calculate the number of dishes to make */
   numDishesToMake = NUM_CUSTOMERS * TRIPS_PER_CUSTOMER;

   for (i=0; i< numDishesToMake; i++) {

   SemaphoreWait(cleanDishes);   // Wait til someone holds out a tray

   // Put some slop onto the tray
   printf("Cook served one, brings total served to %d\n", i+1);
   SemaphoreSignal(fullDishes);  // Ring the bell to start drooling
```

```
   }

   printf("COOK THREAD FINISHED\n");
}



/*
 * Customer
 * --------
 * This is the routine forked by all the Customer threads.  The customer's
 * keeps coming up to the buffer for numTrips iteration to get a plate
 * of yummy food, which the customer quickly eats and then places the
 * dirty dish on the busyboy's cart for the dishwasher to wash.  Each
 * customer asks for the same number of dishes and exits when it got
 * them all.  In this version, the customer doesn't signal a load is
 * ready to be washed until either the min load size is reached, or this
 * is the last customer exiting.
 */
void Customer(int custNum)
{
   int i;

   for (i=0; i< TRIPS_PER_CUSTOMER; i++) {

      SemaphoreWait(fullDishes); // wait til cook serves something
      printf("Customer #%d served on trip %d\n", custNum, i+1);
      SemaphoreWait(dirtyLock);  // acquire lock for global var
      numDirtyDishes++;      // if got min load, tell dishwasher about it.

      // The % below can be a little confusing.  The purpose is to signal
      // the dishwasher every time numDirtyDishes hits MIN_LOAD, MIN_LOAD +
      // MAX_LOAD, MIN_LOAD + 2*MAX_LOAD, etc.  This way, the dishwasher
      // will be signaled precisely the number of times it needs
      // to go through a wash cycle.  Remember, the dishwasher may not
      // respond immediately to the startLoad signal, and the number of
      // dishes can pile up above and beyond MAX_LOAD.  If this happens,
      // the dishwasher needs to be signaled for each time it should
      // wash its dishes.
      if ((numDirtyDishes % MAX_LOAD) == MIN_LOAD)
         SemaphoreSignal(startLoad);

      SemaphoreSignal(dirtyLock);
   }

   SemaphoreWait(activeLock);   // to access numActiveCustomers
   numActiveCustomers--;   // if last customer, trigger final load

   // If last customer, and we haven't already signaled enough times
   // to cover the remaining dishes, trigger the final load.
   if (numActiveCustomers == 0) SemaphoreSignal(startLoad);
   SemaphoreSignal(activeLock);

   printf("CUSTOMER #%d THREAD FINISHED \n", custNum);
}
```

```
/*
 * Dishwasher
 * ----------
 * This is the routine forked by the Dishwasher thread.  The dishwasher's
 * job will to be washed the used dishes. In this version, the dishwasher
 * is a little more complicated.  It waits until a load has been signalled
 * which means either we've got min_load worth of dishes or the last
 * customer has exited, leaving some odd number of dirty dishes.
 * At each receipt of startLoad signal we will wash a full load if we
 * can, or a partial load, it that's all that's left.  We then need to
 * signal for each clean dish that is now available.
 */

void Dishwasher(void)
{
    int i, numThisLoad;
    int numWashed = 0;

    while (numWashed < (NUM_CUSTOMERS * TRIPS_PER_CUSTOMER)) {

   SemaphoreWait(startLoad);     // wait til ready for load

   SemaphoreWait(dirtyLock);     // acquire lock for numDirty
   numThisLoad = MIN(numDirtyDishes, MAX_LOAD); // wash what we can
   numWashed += numThisLoad;
   numDirtyDishes -= numThisLoad;
   SemaphoreSignal(dirtyLock);   // release numDirty lock

   printf("Dishwasher washed %d, brings total washed to %d\n",
          numThisLoad, numWashed);

   for (i = 0; i < numThisLoad; i++)// Signal load of clean dishes
       SemaphoreSignal(cleanDishes);
    }

    printf("DISHWASHER THREAD FINISHED\n");
}
```

```
/*
 * Our main creates a group of customer threads and the single cook
 * and dishwasher threads. The cleanDishes semaphore starts out with
 * all dishes being clean, and the number of full dishes starts out at zero.
 * The startLoad semaphore is initialized to zero (no load is ready yet)
 * and the two variable locks (dirty & active) start available
 * (one thread can acquire the lock).
 */

int main(int argc, char **argv)
{
    int i;
    char name[32];
    bool verbose = (argc == 2 && (strcmp(argv[1], "-v") == 0));

    InitThreadPackage(verbose);

    for (i = 0; i < NUM_CUSTOMERS; i++) {
      sprintf(name, "Customer %d", i);
      ThreadNew(name, Customer, 1, i);
    }
    ThreadNew("Cook", Cook, 0);
    ThreadNew("Dishwasher", Dishwasher, 0);

    cleanDishes = SemaphoreNew("Clean Dishes", NUM_DISHES);
    fullDishes = SemaphoreNew("Full Dishes", 0);
    startLoad = SemaphoreNew("Start Load", 0);
    dirtyLock = SemaphoreNew("Dirty Dishes Lock", 1);
    activeLock = SemaphoreNew("Active Customer Lock", 1);

    numActiveCustomers = NUM_CUSTOMERS;
    numDirtyDishes = 0;

    RunAllThreads();
    SemaphoreFree(cleanDishes);
    SemaphoreFree(fullDishes);
    SemaphoreFree(startLoad);
    SemaphoreFree(dirtyLock);
    SemaphoreFree(activeLock);

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

Example 2: Locks

This example illustrates the use of the Semaphore data type to build a more sophisticated data type known as a lock.

Suppose we are given the task to design a linked list data structure that can be accessed concurrently by multiple threads. The linked list will operate just like a normal linked list except that all its crucial operations (like `ListInsert` and `ListDelete`) need to be designed so that only one thread at a time can access and modify the internals of the list data structure. In other words, a single thread must be given mutually exclusive access to the internals of the list when performing operations on it.

One way we might choose to implement mutual exclusion on list operations is by associating a binary semaphore `listLock` with the list. At the beginning of each operation (like `ListInsert`), we call `SemaphoreWait(listLock)` to get mutually exclusive access to the list. Before exiting the operation, we call `SemaphoreSignal(listLock)`. A somewhat careless implementation of `ListInsert` follows:

```
// Structure: Node
// The basic building block for a linked list. Contains
// a pointer to the next node in the list as well as
// the data for this node.

typedef struct _Node {
   struct _Node * next;
   int listData;
} Node;

// Structure: CList
// A CList is a linked list that allows concurrent access
// by multiple threads. The CList data structure contains
// a pointer to the first node in the linked list, as well
// as a semaphore which is used to control access to the
// list.

typedef struct {
   Node* head;
   Semaphore listLock;
} CList.

// Operations like ListNew, ListDelete, etc. are omitted.

// ListInsert
// Inserts a node at the end of the list. If the CList is
// empty the head pointer is NULL. Note that there is a
// BUG in this code.

void ListInsert(CList * list, Node * newNode)
{
   Node *prev = NULL, *cur;
   // make sure we are the only thread that is accessing
   // the list
   SemaphoreWait(list->listLock);
```

```
   // Iterate to the end of the list, updating prev and cur
   for (cur = list->head; cur != NULL; cur = cur->next)
      prev = cur;

   if (prev == NULL) { // list was previously empty
      list->head = newNode;
      SemaphoreSignal(list->listLock);
   }
   else // append new Node to tail of list
      prev->next = newNode;

   // Let others have access to the list.
   SemaphoreSignal(list->listLock);
}
```

The obvious bug in `ListInsert` is that `SemaphoreSignal` is called twice when `prev == NULL`. The danger of this is that the value of the `listLock` will be incremented twice and the next *two* requests to access listLock will be granted, meaning that the mutual exclusion could be compromised.

This careless coding illustrates a general problem with semaphores. Unmatched `SemaphoreWait`/`SemaphoreSignal` calls can cause errors which are hard to detect. Ideally, given the task of making global data concurrent (like the CList), we want a Semaphore-like object that will catch and assert the bug in the previous example. Thus we introduce the concept of a *lock*. A lock is a wrapper over a Semaphore which ensures that the value of the Semaphore is strictly binary (0 or 1) and asserts if the value is otherwise. A lock has two operations: `LockAcquire` and `LockRelease`, similar to `SemaphoreWait` and `SemaphoreSignal`.

A simple but seriously flawed implementation of a Lock is as follows:

```
typedef struct {
   Semaphore dataAccess; // Protects var with mutual exclusion
   bool inUse; // Is lock already in use?
} LockStruct;

// LockAcquire
// Implements similar functionality as SemaphoreWait,
// except that the value of the lock is guaranteed
// to be either 0 or 1. This implementation of LockAcquire
// does busy waiting until the lock is no longer in use,
// at which point it takes access to the lock. This example is for
// illustration purposes only as busy waiting takes up unnecessary CPU time
// and a better solution will be found.

void LockAcquire(LockStruct * lock)
{
   bool haveLock = false;

   /* Continue looping until we get a lock */
   while(!haveLock) {
      SemaphoreWait(lock->dataAccess);
      // If it's not already in use, grab it
```

```
      if (!lock->inUse) {
         lock->inUse = true;
         haveLock = true;
      }
      SemaphoreSignal(lock->dataAccess);
   }
}

// LockRelease
// Implements similar functionality as SemaphoreSignal,
// except that the value of the lock is guaranteed
// to be either 0 or 1. If prior to LockRelease a LockAcquire
// has not been called, the assert statement will be raised.
void LockRelease(LockStruct * lock)
{
   SemaphoreAcquire(lock->dataAccess);
   // The lock should have been in use, otherwise why
   // was LockRelease called?
   assert(lock->inUse);
   lock->inUse = false;
   SemaphoreRelease(lock->dataAccess);
}
```

The flaw in the above implementation is with `LockAcquire`. The while loop executes again and again until the `inUse` variable becomes false. This type of behavior is known as *busy waiting*, and is clearly a problem in a real multiprogramming system where a single CPU is shared among many processes — processes that are busy waiting will unnecessarily utilize system resources.

The solution to the problem is to introduce another semaphore, `lockAvailable`, which is signaled by `LockRelease` whenever `inUse` is set to false. Here is the correct implementation:

```
typedef struct {
   Semaphore dataAccess;       // Protects var with mutual exclusion
   bool inUse;           // Is lock already in use?
   Semaphore lockAvailable; // Reflects the value of inUse
} LockStruct;

// LockAcquire
// Implements similar functionality as SemaphoreWait,
// except that the value of the lock is guaranteed
// to be either 0 or 1. This implementation of LockAcquire
// avoids busy waiting by waiting instead on the lockAvailable
// Semaphore, which is signaled by LockRelease when the lock
// becomes unused.

void LockAcquire(LockStruct * lock)
{
   SemaphoreWait(lock->lockAvailable);

   SemaphoreWait(lock->dataAccess);
   assert(!lock->inUse);
   lock->inUse = true;
   SemaphoreSignal(lock->dataAccess);
```

```
}

// LockRelease
// Implements similar functionality as SemaphoreSignal,
// except that the value of the lock is guaranteed
// to be either 0 or 1. If prior to LockRelease a LockAcquire
// has not been called, the assert statement will be raised.
// The lockAvailable semaphore is signaled to indicate
// the lock is now available.
void LockRelease(LockStruct * lock)
{
   SemaphoreWait(lock->dataAccess);

   assert(lock->inUse);
   lock->inUse = false;
   SemaphoreSignal(lock->lockAvailable);
   SemaphoreSignal(lock->dataAccess);
}
```

Another Puzzle to Ponder

Here's another funky example that involves a counting contest between 2 threads. Imagine a global integer counter that starts out with the value 0, and is protected by a binary semaphore. There are 2 threads in this scenario, where one loops through trying to increment the counter to 10 (let's call this thread Lisa), and the other loops through trying to decrement the counter to -10 (let's call this one Bart!). Here is some actual code that implements this scenario, followed by some questions you can think about:

```
/*
 * race.c
 * -------
 * This file implements a simple program demonstrating a race between two
 * "players".  One player, the Incrementer, wants to increase a global
 * variable (initially set to 0) to GOAL.  The opponent (Decrementer)
 * wants to decrease that same variable to -GOAL.
 *
 * Jon Goldberg -- Spring, 1996
 */

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

#define GOAL 10
#define SLEEP_DELAY 10

static Semaphore mutex;
static int gCounter;

/* Implements a generic player, who will either decrement or increment the
 * global counter according to the parameter isIncrement.
 */
void Player(bool isIncrement, bool wantSleep)
{
  int lastSet;
  int adjust = 1;
```

```
  if (! isIncrement)
    adjust = -adjust;

  while (true) {
    SemaphoreWait(mutex);

    if ((gCounter == GOAL) || (gCounter == -GOAL)) {
      SemaphoreSignal(mutex);
      break;
    }
    gCounter += adjust;
    lastSet = gCounter;

    SemaphoreSignal(mutex);

    printf("%s set counter to '%d'\n", ThreadName(), lastSet);

    if (wantSleep)
      ThreadSleep(SLEEP_DELAY);
  }
  printf("%s THREAD FINISHED\n", ThreadName());
}

/*
 * Our main creates two threads (one for each player) and the mutex binary
 * semaphore.  It also initializes the global counter to 0
 */
int main(int argc, char **argv)
{
    bool verbose = (argc == 2 && (strcmp(argv[1], "-v") == 0));

    InitThreadPackage(verbose);

    ThreadNew("INCREMENT",Player, 2, true , true);
    ThreadNew("DECREMENT",Player, 2, false, true);

    mutex = SemaphoreNew("Mutex", 1);
    gCounter = 0;

    RunAllThreads();
    SemaphoreFree(mutex);
    printf("All done!\n");
    return 0;
}
```

Questions about the race:

1. (a) What are the possible outcomes or types of outcomes?
   (b) Under what circumstance would each type of outcome occur?
   (c) What would be the impact of initializing `wantSleep` to `false`?

2. (a) What are the ways one of the processes might cheat?
   (b) What would be the outcome in each case?

Final Words of Advice: What's Hanging?

Here are some tips when debugging a concurrent program. When your program hangs, chances are that it is hanging on semaphore that is never getting that awaited signal. Some things to look out for:

1. Make sure all your semaphores are properly initialized (i.e. start out with the right value)

2. For each `SemaphoreWait()` call there is a corresponding `SemaphoreSignal()`. This is analogous to how you should use `malloc()` and `free()`.

3. Watch out for the order in which semaphore locks are acquired and released. Again, this is somewhat analogous to the use of `malloc()` and `free()`.

4. Watch out for early loop termination. Frequently, even though it looks like you have the matching numbers of waits and signals, there is some early termination condition which causes the loop to be terminated before the full number of iterations, whereas some other function is still waiting on a semaphore for a signal that will never come.

5. Make use of the debugging functions like `ThreadName` and `ListAllThreads` provided by the thread package.