Part I – 20 points each

1. Suppose proposed critical section entry/exit routines are defined as follows using a shared variable locked that is initialized to zero (unlocked):

   ```
   entry(int *lock) {
        // assume TSL generates an assembler
        // test and set lock instruction
        while (TSL(lock))
             ;
   }

   exit(int *lock) {
        *lock = 0;  // reset lock
   }
   ```

   Does this implement a critical section? Justify your answer.

2. A robot relay race requires 3 robots to run 1/3 of a track each. The first robot starts when a sensor indicates that a gun has fired. The second robot is not allowed to start until the first robot has reached the 1/3 point of the track, and the third is not allowed to start until the second has reached the 2/3 point. Using Dijkstra's counting semaphores, write pseudocode for each robot to implement barriers that would allow the race to be completed without any false starts. Be sure to initialize your semaphores to an appropriate value. Use the following functions:

   WaitForGun() – A function that returns once the gun has been fired (no need to use a barrier here).
   FollowTrack() – Function that moves the robot one meter along the track. It returns True if the robot has reached the finish with respect to its 1/3 of the track, or False if there is more track to run.
   Robot$_1$(), Robot$_2$(), Robot$_3$() or Robot(idx) – Provide functions for robots 1, 2, and 3 either separately or as a common function with the robot number passed in as idx.

   Semaphores can be initialized to an integer value and have only two functions that can be used on them: up() and down().

3. Write pseudocode to implement a remote procedure call (RPC) to another process listening on named message queue "rpc". The RPC implements a an interface to

a flood control dam and takes two arguments that you must package into a buffer that is to be sent and unpacked on the other side: water level and chance of rain, both of which are obtained by calls to sensorHoHLevel() and predictionRain(). On the remote side, a message processing loop takes the arguments passed through the message interface and calls floodControl(walter level, chance of rain) and returns a number indicating the percent to which the flood gates should be opened. This interface should run every 3 m. Assume the following functions which are loosely inspired by a POSIX specification with several simplifications to the interface and an assumption of guaranteed delivery:

message queue functions

mq_type mqOpen(char *qname, char *mode, int maxnummsg) – Open named message queue qname in mode "r" read, "w" write, or "rw" read-write. Variable maxnummsg indicates the message capacity of the channel in number of messages (not bytes).

void * mqReceive(mq_type queue) – Blocks until a message is available at which point the address of the message is returned.

void mqSend(mq_type queue, void *msg, size_t msgSize) – Send the msgSize bytes contained in the buffer pointed to by msg to the specified message queue.

You may also assume void sleep(int N) causes the process to sleep for N seconds.

4. An adaptive mutex is a semaphore that makes a decision as to whether or not to busy wait depending upon the state of other processes in a multi-processor system. When a process attempts to wait on a semaphore, the following is done (for simplicity, we consider a mutex between two processes only):

```
wait (Semaphore S) {
     S.value = S.value - 1;
     if (S.value < 0) {
      /* Some other process P holds the semaphore */
      P = process in critical region for semaphore S;
      if (state(P) == running) {
         while (S.value < 0)           /* busy wait */
             no-op;
      } else {
         add calling process to wait list for S;
         block calling process;
      }
     }
}
```

What is the rationale for providing such a semaphore?

Part II – POSIX programming assignment 120 points

Suppose that Lucy and Ethel have gone to work for the Mizzo candy factory. Mizzo produces two types of candy: crunchy frog bites and everlasting escargot suckers. Unlike their last job, Mizzo has automatic flow control on their assembly line. No more than 10 candies are on the conveyer belt at any given time.

Crunchy frog bites tend to be more expensive than escargot suckers, so Mizzo has implemented a policy to prevent putting too many frog bites in the same box. No more than 3 frog bites can be on the conveyer belt at any given time. (Candies are taken off the line in the order that they are produced.)

Write a program using POSIX unnamed semaphores and POSIX threads to simulate this multiple producer and multiple consumer problem. POSIX unnamed semaphores are covered in the frequently asked questions section of the course web site.

Your program must meet the following design criteria:

1. The program should take the following optional command line arguments:

   -E N Specifies the number of milliseconds N that the Ethel consumer requires to put a candy in the box and should be invoked each time Ethel removes a candy regardless of the candy type.

   -L N Similar argument for the Lucy consumer.

   -f N Specifies the number of milliseconds between the production of each crunchy frog bite.

   -e N Specifies the number of milliseconds between the production of each everlasting escargot sucker.

   If an argument is not given for any one of the threads, that thread should incur no delay. The class FAQ explains command line argument parsing and how to cause a thread to sleep for a given interval (remember from the last assignment that using getopt can make parsing much easier). You need not check for errors when sleeping.

2. Your program should written using multiple files that have some type of logical coherency (e.g. producer, consumer, belt, etc.). Write your Makefile (required with all programs) early, this will permit you to not have to worry about which files have changed since the last time you compiled. The Makefile should generate program **mizzo** (lower case) and be in directory **cs570/a05** on your edoras account.

3. *Do not use global variables* to communicate information to your threads. Pass in data structures.
4. Each candy generator should be written as a separate thread. The consumer processes (Lucy & Ethel) must share common code but must be executed as separate threads. It is also possible to share common code for the producers, but not required.
5. One of the elements of the data structure that you pass to your consumer threads should be the consumer thread name (Lucy or Ethel), this will allow you to print messages indicating whether Lucy or Ethel did the work.
6. Maintain the ordering of the candy production and consumption. Candies are removed in first-in first-out order.
7. Your producers should stop production once 100 candies are produced. After all 100 candies are consumed, the program should exit. Descriptive output should be produced each time a candy is added or removed from the conveyer belt. When the day's candy production is complete, you should print out how many of each type of candy was produced and how many of each type were processed by each of the consumer threads.

Sample output (order depends on thread scheduler and parameters):

```
Belt: 1 frogs + 0 escargots = 1. produced: 1    Added crunchy frog bite.
Belt: 1 frogs + 1 escargots = 2. produced: 2    Added escargot sucker.
Belt: 0 frogs + 1 escargots = 1. produced: 2    Ethel consumed crunchy frog bite.
Belt: 0 frogs + 0 escargots = 0. produced: 2    Lucy consumed escargot sucker.
Belt: 0 frogs + 1 escargots = 1. produced: 3    Added escargot sucker.
...

PRODUCTION REPORT
----------------------------------------
crunchy frog bite producer generated 46 candies
escargot sucker producer generated 54 candies
Lucy consumed 24 crunchy frog bites + 30 escargot suckers = 54
Ethel consumed 22 crunchy frog bites + 24 escargot suckers = 46
```

If you are having difficulties understanding semaphores, my suggestion is to write this project in stages. First, make a single producer and consumer function on a generic candy type function. Then, add multiple producers and consumers. Finally, introduce the multiple types of candy. If you only get the multiple producer and consumer threads working with a single candy type (and meet the other criteria, e.g. separate compilation, etc.), you will earn a score of right track.

HINT: One of the difficult problems for students is how to stop the program. Imagine that the Lucy thread consumes the 100th candy. The Ethel thread could be asleep, and thus never able to exit. The trick here is to use a barrier in the main thread that is signaled by the consumer that consumed the last candy. The main thread should block until consumption is complete.

What to turn in

- Paper copy of your work including the program, Makefile, output with the parameters –f 600 –e 400 –L 300 –E 500. Pair programmers should turn in a single package containing the questions from part I (answered separately) and a single copy of the program.
- All students must fill out and sign either the single or pair affidavit and attach it to your work.