Assignment 5

Part I - Answer the following questions (20 points each)

1. Suppose 4 pages are in memory and are accessed at the following times:

```
page 0:  3, 33, 97
page 1: 24, 55
page 2: 25, 54
page 3: 18, 28, 36
```

   Assuming a least recently used with aging page replacement algorithm with a 20 ms history update and 3 history bits show the history data structure for these pages at time 20, 40, 60, 80, and 100 ms and indicate what page would be replaced if a page fault occurred before the next history update.

2. Using Dijkstra's counting semaphores, modify the following pseudocode so that operations on shared variables are atomic. Keep your critical regions as small as possible.

```
main() {
        int x;
        for (j=0; j <  NumberThreads; j++) {
                start new thread executing foobar(&x);
        }
}

foobar(int *x) {
        y = 2 * (*x);
        print y;
        z = 2 * y;
        *x = *x / 2;
}
```

3. The semaphore synchronization below is intended to ensure that function foobar does not complete before all worker threads have finished (successfully or unsuccessfully). Will the synchronization always function as intended for this code? Justify your answer.

```
void foobar()   /* main code */
{
      semaphore barrier = NUM_WORKERS

      /* other code ... */

      try {
            for (i = 0; i < NUM_WORKERS; i++)
                  start_thread(worker, barrier)
      } catch any exception {
            abort program, killing all child threads
      }
      for (i = 0; i < NUM_WORKERS; i++)
            barrier.down()
      print "all done"

}

void worker_thread(semaphore s) {
      try {
            /* commands to do work... */
      } catch any exception {
            print message to error log
            s.up();
            exit thread
      }
      s.up();
}
```

Part II: 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. You would simulate this time to consume a product (put a candy in the box) by putting the consumer thread to sleep for N milliseconds. Other consumer and producer threads (Lucy, frog bite, and escargot sucker) are handled similarly.

   -L N Similar argument for the Lucy consumer.

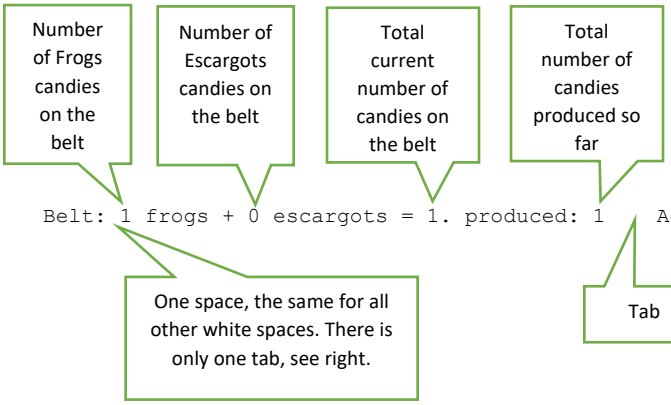
   -f N Specifies the number of milliseconds required to produce each crunchy frog bite.

   -e N Specifies the number of milliseconds required to produce each everlasting escargot sucker.

2. 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.
3. Your program should be 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).
4. Do not use global variables to communicate information to your threads. Pass information through data structures.
5. Each candy generator should be written as a separate thread. When creating these threads, create the crunchy frog bite generator then the everlasting escargot sucker. 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. In consumer thread creation, Lucy should be created before Ethel.
6. 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.

7. Maintain the ordering of the candy production and consumption. Candies are removed in first-in first-out order.
8. 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**. Your output format should be as specified in the sample output below. 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 + 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

1. A5 code – Submit your code and Makefile. Do not submit object files or programs.
2. A5 Question & Code Listing – Submit your A5 document package. This is **a single file** containing:
   - Single or pair affidavit – No digital signature is required, you may type your name.
   - Answers to Part I
   - Your formatted code that the grader can look at and provide feedback. There is no easy way to mark up your code files and return them to you. Ensure that the code is readable and properly indented. If your code wraps, either learn how to format the page in landscape mode or stick to 80 columns (preferred). Most editors will let you set a guide that shows the location of the 80 column margin which will let you write your code such that it will print nicely.
   - Your code output.