Assignment 4

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: 12, 55
page 2: 25, 54, 62
page 3: 18, 28, 36
```

   Assuming a not frequently 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. Use the following table to work out your solution.

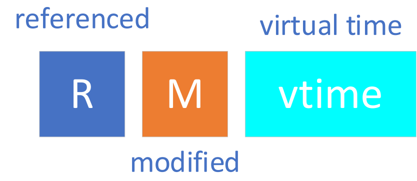
Time	Page	History (3 bit string)	Referenced	Victim
20	0			
	1			
	2			
	3			
40	0			
	1			
	2			
	3			
60	0			
	1			
	2			
	3			
80	0			
	1			
	2			
	3			
100	0			
	1			
	2			
	3			

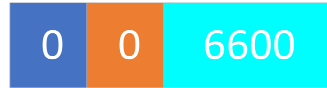
2. Why is instruction backup required for paging systems?

3. The semaphore synchronization pseudocode 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_thread, 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();
}
```

4. The working set clock algorithm has been invoked on the clock below to service a page fault. The current process virtual time is 7000 and the threshold τ=500. Draw a new clock that shows the results of running the working set clock algorithm. Indicate the victim frame with a V beside the frame/page and update the page bits and virtual time to reflect the newly assigned page being assigned to the frame. Indicate any pages scheduled for writing with a W beside the page. Update the status of the reference bits of any pages as needed. Draw the clock hand to point to the page where the next clock walk would start when the next page fault happens.


referenced
virtual time
R
M
vtime
modified


0
0
6600

1
1
6950

start from
here

0
1
5000

0
0
6400

1
0
6200

1
1
6800

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. This delay occurs each time Ethel removes a candy regardless of the candy type. You cab 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.

   a. The Makefile should generate program **mizzo** (lower case).
   b. Compilation within the Makefile should use the -g compile flag to generate gdb debugging information.
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. Maintain the ordering of the candy production and consumption. Candies are removed in first-in first-out order.
7. Each time the belt is mutated (addition or removal), a message should be printed indicating which thread performed the action and the current state. Functions in **io.c** will let you produce output in a standard manner. Use **io_add_type** and **io_remove_type** to print messages.
8. Your producers should stop production once 100 candies are produced. After all 100 candies are consumed, the program should exit. 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, use **io_production_report**. Functions are available on the canvas assignment page.

Sample output from: ./mizzo -f 5 -e 15 -E 35 -L 20

Order is nondeterministic and depends on thread scheduler and parameters:

```
Belt: 1 CFB + 0 EES = 1. Added crunchy frog bite. Produced: 1 CFB + 0 EES = 1 in 0.000 s.
Belt: 0 CFB + 0 EES = 0. Lucy consumed crunchy frog bite.  Lucy totals: 1 CFB + 0 EES = 1 consumed in
   0.000 s.
Belt: 1 CFB + 0 EES = 1. Added crunchy frog bite. Produced: 2 CFB + 0 EES = 2 in 0.005 s.
Belt: 0 CFB + 0 EES = 0. Ethel consumed crunchy frog bite.  Ethel totals: 1 CFB + 0 EES = 1 consumed in
   0.005 s.
Belt: 0 CFB + 1 EES = 1. Added everlasting escargot sucker. Produced: 2 CFB + 1 EES = 3 in 0.010 s.
… lots of lines removed …
Belt: 0 CFB + 1 EES = 1. Lucy consumed everlasting escargot sucker.  Lucy totals: 23 CFB + 40 EES = 63
   consumed in 1.245 s.
Belt: 0 CFB + 0 EES = 0. Lucy consumed everlasting escargot sucker.  Lucy totals: 23 CFB + 41 EES = 64
   consumed in 1.265 s.

PRODUCTION REPORT
----------------------------------------
crunchy frog bite producer generated 34 candies
everlasting escargot sucker producer generated 66 candies
Lucy consumed 23 CFB + 41 EES = 64 total
Ethel consumed 11 CFB + 25 EES = 36 total
Elapsed time 1.268 s
```

If you are having difficulties understanding semaphores, we suggest that you 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 and kill the child threads (or simply let the OS kill them when the program exits).