Spring 2017 **EECS 338 Assignment 5** **Due:** March 29, 2017 (Wed)
G. Ozsoyoglu **Linux System V Shared Memory and Semaphore Programming**

100 points

Baboon Crossing Problem (semaphore-based process synchronization). There is a deep canyon somewhere in the Katavi National Park, Tanzania, and a single rope that spans the canyon from point A to point B. Baboons can cross the canyon by swinging hand-over-hand on the rope, but if two baboons going in opposite directions meet in the middle, they fight until one or both drop to their deaths. Furthermore, the rope is only strong enough to hold 5 baboons. If there are more baboons on the rope at the same time, it breaks.
In our environment, baboons (i.e., baboon processes) have become smarter and now use semaphores for peacefully using the crossing, instead of fighting and killing each other.
Assuming that the baboons can use semaphores, we would like to design a rope crossing synchronization scheme with the following properties:

- *Ordered crossing.* Baboons arriving point A (B) line up, and cross the rope to B (A) in the order they arrive.
- *Rope load.* There cannot be more than five baboons on the rope: the rope cannot handle the load.
- *Crossing guarantee.* Once a baboon process begins to cross the rope, it is guaranteed to get to the other side without running into another baboon going the other way. (Remember, baboons are smart now).
- *Streaming guarantee when only one side has baboons waiting to cross.* If multiple baboons arrive to point A, and, there are no baboons at point B, they can all cross in sequence, subject to the rope load and fairness restrictions.
- *Fairness (no starvation).* A continuing stream of baboons crossing in one direction should not bar baboons from crossing the other way indefinitely. So, after every 10 baboons complete their crossing from A to B (or vice versa), if baboons are waiting to cross from B to A, the crossing direction must switch into a B-to-A crossing.

Write a UNIX System V semaphore/shared memory-based program for the Baboon Crossing problem. Explain your code, and explicitly specify any assumptions you make about the model.

Requirements and Hints:

- Explain your code, and explicitly specify any assumptions you make about the model.
- To test your code, you can fork your baboon processes from your *main()* in a preset manner. This has the advantage that you can debug your code under specific circumstances. Or, you can fork baboon processes from *main()* by sleeping for randomly chosen time durations (via *rand()* or *srand()*) and awakening and forking baboon processes.
- Test your code by having multiple runs, with each run having many (at least 20) baboon processes. Needless to say, your output should have printouts, helping the TAs to understand and grade your code.
- Implement the semaphore-based algorithm given to you in the solutions for assignment #4.
- Process communication among processes are to be implemented via Unix System V shared memory and semaphore primitives, covered in recitations.
- You can directly use Linux System V primitives in C. Or, you can use *wait()* and *signal()* implementations (via System V semaphore commands) as listed at http://eecs-002.case.edu/338.S17/exampleprograms.html
- Use caution when casting. Do not cast *semid* and *shmid* to a different type. *shmat* gives you a *void** value (i.e., a pointer to a memory region with no associated type). You will want to cast *shmat* to whatever data type you want, such as in https://github.com/cwru-eecs338/smokers/blob/master/smoker.c.
- Make sure that your shared variables are mutually exclusively modified and/or accessed.
- Make sure to use *sleep()* calls to slow down, and, thus, to observe the behavior of customer processes.

Run your program in the script environment, just like in previous assignments. On the due date, submit your code, MakeFile, and (multiple) program outputs (that illustrate the correctness of your code) to the blackboard **as one single zip file.** Remember to error-check all system calls: check return values for success, and use *perror* when possible on failure.