Fall 2015 **EECS 338 Assignment 4** **Due:** Oct 29, 2015

G. Ozsoyoglu *Linux System V Shared memory and Semaphores*

Savings Account Problem; 100 points

In this assignment you will use semaphore and shared memory constructs of Linux system V to implement the Savings Account Problem**:**

> *A savings account is shared by several people (i.e., processes). Each person may deposit or withdraw funds from the account. The current balance in the account is the sum of all deposits to date minus the sum of all withdrawals to date.*
>
> *The balance must never become negative, and withdrawals are serviced First-Come-First-Serve (FCFS) with one twist: those who want to withdraw money from the account are desperate to get their money; so, if the balance is not sufficient, they wait (in a FIFO order) until somebody else deposits enough money for them to make their withdrawals.*
>
> *Note that those waiting for others to deposit (so that they can withdraw) are serviced strictly FCFS. For example, suppose the current balance is 200 dollars, and customer A is waiting to withdraw 300 dollars. Assume another customer, say B, arrives with the request of withdrawing 200 dollars. Customer B must wait until customer A is serviced.*

- We have studied in the class a monitor-based algorithm for the Savings Account problem. In this assignment, your task is to code and run a C program that uses Linux System V semaphore/shared memory system calls for the Savings Account problem. In the next page, you are given a semaphore-based algorithm for the Savings Account problem--You may or may not base your implementation on this algorithm (study it first).

Requirements and Hints:

- Explain your code, and explicitly specify any assumptions you make about the model.
- To test your code, you can fork your customer 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 customer processes from *main()* by sleeping for randomly chosen time durations (via *rand()* or *srand())* and awakening and forking a customer process.
- Test your code by having multiple runs, with each run having many (at least 10) customer processes. Needless to say, your output should have printouts, helping the TAs to understand and grade your code.
- Process communication among processes is to be implemented via *Linux 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://art.case.edu/338.F15/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.
- Do not forget to conform to the assignment grading policy requirements listed at http://art.case.edu/338.F15/grpolicy.html

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.

```
binary semaphore mutex := 1;
nonbinary semaphore wlist := 0; // Customers wait FCFS on this semaphore to withdraw.
int wcount := 0;                                  // The number of waiting withdrawal customers on wlist
int balance := 500;
linked-list LIST := NULL;      // The list containing the withdrawal requests of
                                                  // waiting customers.
// Linked-list manipulation procedures:
void AddEndOfList (linked-list A, int Val);       // Adds an element with value Val to the end of list A.
void DeleteFirstRequest (linked-list A);          // Deletes the first element in list A.
int FirstRequestAmount (linked-list A);           //Returns the value in the first element in list A.
```

Depositing Customer:

```
      // Assume that the local variable deposit (int) contains the amount to be deposited.
wait (mutex);
balance := balance + deposit;
if (wcount = 0) signal (mutex)                    // no withdrawal requests at this time.
else if (FirstRequestAmount (LIST) > balance ) signal (mutex); // Still not enough balance for 1st waiting
                                                  // withdrawal request.
     else signal (wlist);                         // Release the earliest waiting customer. Note that mutex is released
                                                  // by the withdrawal customer—to avoid race conditions.
// Deposit has taken place.
```

Withdrawing Customer:

```
            // Assume that the local variable withdraw (int) contains the amount to be withdrawn.
wait (mutex);
if (wcount = 0 and balance > withdraw)   // Enough balance to withdraw.
       {balance := balance – withdraw; signal (mutex)}
else {wcount := wcount + 1;              // Either other withdrawal requests are waiting or not enough balance.
      AddEndOf List (LIST, withdraw);
       signal (mutex);
       wait (wlist);                                  // Start waiting for a deposit.
       balance := balance – FirstRequestAmount (LIST);  // Withdraw.
       DeleteFirstRequest (LIST);          // Remove own request from the waiting list.
       wcount := wcount – 1;
       if (wcount>1 and (FirstRequestAmount (LIST))<balance)) signal(wlist)
       else signal (mutex)}                // This signal() is paired with the depositing customer's wait(mutex).
// Withdrawal is completed.
```