Programming Assignment 1: Managing User-Level Threads
Due date: November 1st, 2006.

The goal of this assignment is to manage four user threads in a single process.

Problem statement

You will solve the producer/consumer problem by managing four user level threads within a single process. This means you will have to perform context switching and scheduling yourself. Since this is a classic IPC problem, you will also need to simulate a function that would allow the buffer insertion or deletion to be done atomically. You can pick the quantum time yourself. Typically, an acceptable value is 100 or 200 milliseconds. The producer will insert an ASCII character into a finite buffer of size 3,000,000 bytes.

Suggestions:

1) You will need to create two functions one for producer and another one for consumer. Each function will be represented by two user-level threads. The main function is the scheduler.
2) We will use FIFO scheduling so in this case, you will simply select the head thread in the runnable queue each time a quantum is reached.
3) When a quantum is reached, you have to stop the current running function and save its stack environment and register values.
4) Once you have successfully swapped out the contents of a thread, you will then need to bring in the contents of another thread.
5) You will need three semaphores for this problem. You have to create your own semaphore that would allow all threads to be synchronized and provide mutual exclusion. **NOTICE: I am not asking you to use semaphore mechanisms provided by Linux.** What I am asking here is for you to create user level semaphore mechanism that will work with your assignment**. Each semaphore will need to have its own blocking queue. You semaphore mechanism will only be visible within your process.** Think about way to synchronize all threads so that swapping will not take place in the middle of a critical section. The *up* and *down* operation should also work as described in the book. Here are a few APIs that you should have:
   - *int * create_sem(void)*---this function returns the starting address of a storage space for semaphore variable (*handle*). You can use *malloc()* to create this space.
   - *void sem_up (int * handle)*---performs *up* operation.
   - *void sem_down (int * handle)*---performs *down* operation.
   - *void sem_delete (int * handle)*---deletes the storage space for the semaphore.
6) When threads are waiting on semaphores (e.g. consumer thread waiting on an empty buffer or producer thread waiting on a full buffer), it should not be in the runnable queue. Therefore, you should find a simple way to maintain the thread

states. When the semaphore is released, you can move all threads waiting on that semaphore into the runnable queue.

Summary:

The major components would include:

- a circular buffer
- a semaphore mechanism (you will need to instantiate 3 instances)
    - with its own blocking queue
    - initialization up, down, delete functions
- four user-level threads based on Lab 3 (2 producer and 2 consumer threads)
- a timer event for scheduling
- a runnable queue, etc.

Submission procedure:

1. Prepare a project report that includes
    i. Your name
    ii. A read-me section on how to compile and run the program
    iii. The difficulties faced in the project
    iv. Your approach to the problem (design document)
    v. The number of hours spend on the assignment
    vi. All source code with meaningful comments
2. Create a tar-ball that contains all the source files and the project report.
3. Submit your tar-ball using hand-in.

Grading Criteria:

Correctness: 65%
Readable and insightful comments: 20%
(hint: at the beginning, provide detailed information of the function. Information can be "what it does and how you build it?")
Completeness of report: 15%