Homework 1: Implementation of Monitors (100 points)

Due: October 11, 2006 by 5:30 pm

1 Producer/Consumer with Semaphores (35 pts)

The goal of this problem is to solve a producer/consumer problem using semaphores. You will use the pthread package to create 4 producer threads and 4 consumer threads. Each producer thread attempts to insert character 'X' into a circular buffer of size 3,000,000 characters. Each consumer thread tries to removes a character from the buffer. You will perform your work in *osage.unl.edu*. Use POSIX semaphore (*sem_init, sem_get, sem_post*, etc.). The pseudo-code is given in Figure 1.

2 Producer/Consumer with Monitors (65 pts)

The goal of this problem is to create your own high-level library (monitor) to provide synchronization support. You should leverage on your experience gained during your homework assignment 2 and question 1 in which you have used semaphores to prevent race conditions from occurring. You will perform your work in *osage.unl.edu*. You will also use the pthread package to solve a producer/consumer problem. You will create 6 producer threads and 6 consumer threads. Each producer thread attempts to insert character 'X' into a circular buffer of size 3,000,000 characters. Each consumer thread tries to remove a character from the buffer. Refer to Figure 2 for the pseudo-code. Please follow the implementation guideline below:

1. You will create a new variable type called *condition variable*(*CV*). Basically, condition variables are used to delay processes or threads that cannot continue executing due to specific monitor state (e.g. full buffer). They are also used to awaken delayed processes or threads when the conditions are satisfiable. You will create a new structure call *cond*. The structure consists of an integer variable that indicates the number of processes blocked on a condition variable and a semaphore that is used to suspend threads. There are three operations that can be performed on the CV. They are:

   (a) count(cv)—returns the number of processes blocked on the cv.

   (b) wait(cv)—suspends a process on the cv. It also causes the process to relinquish exclusive access to the monitor.

   (c) signal(cv)—unblocks one process suspended on the cv.

   Notice, you **are not allowed** to use existing condition variables such as one from pthread library (*pthread_cond_init*).

2. The function to create a monitor will be `mon_init` in which the necessary data structure(s) will be created in an arena. The function then returns the reference to the arena if successful or zero if failed. The arena should consist of i) a semaphore to regulate entry and exit to the monitor, ii) all the necessary condition variables (you will need two in this case, *not_full* and *not_empty*), iii) a circular buffer needed to store the characters generated by the producers in the arena, and iv) an integer variable to record the number of items in the buffer. The arena must be created in a memory area that is visible to all threads.

3. Function `mon_enter` performs the down operation on the semaphore in the arena.

4. Function `mon_exit` performs an up operation on the semaphore in the arena.
5. Function `wait` suspends the executing thread on a specified condition variable.
6. Function `signal` awakens the blocked thread at the head of the queue of the specified condition variable.
7. Function `mon_insert` inserts a character into the buffer. If the buffer is full, invoke `wait` on the condition variable *not_full*. It also invokes `signal` on condition variable *not_empty*.
8. Function `mon_remove` removes a character from the buffer. If the buffer is empty, invoke `wait` on the condition variable *not_empty*. It also invokes `signal` on condition variable *not_full*.

All these functions will be implemented in a separate C file. Thus, you should at least have two C source files, *monitor.c* and *pro_con.c*. You can compile *monitor.c* using -c flag (e.g. gcc -c monitor.c). This will give you the object file (monitor.o) that can be linked to your *pro_con.c* (gcc pro_con.c monitor.o).

3 Submission procedure

Create a zip file that has all your solutions and submit through hand-in. The step to create proper directory structure is as follows:

1. Create a directory called *lastname_lab2* (replace *lastname* with your lastname).
2. Create subdirectories: *prob1 and prob2* under *lastname_lab2*.
3. Place your solutions in the proper directory. Provide README.txt file for each problem. Each README file should specify:
   - Specific instructions on how to test your solution.
   - How much time you spent on that particular problem.
   - The level of challenge from 0 to 5 (5 is most difficult).
   - How much prior knowledge do you have to attack the problem.
4. Once all solutions are properly stored, zip the folder *lastname_lab2* and submit the zip file through handin.

```
#define N 3000000
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void main(void)
{
    //create four producer threads
    //create four consumer threads
}

void producer(void)
{
    while(1)
    {
        down(&empty);
        down(&mutex);
        insert('X');
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    while(1)
    {
        down(&full);
        down(&mutex);
        remove();
        up(&mutex);
        up(&empty);
    }
}
```

Figure 1: Pseudo-code for producer/consumer problem using semaphores

```
#define N 3000000
// arena
semaphore mutex = 1;
cond empty;
cond full;
int count;
char buf[N];
// the five variables above should be created as arena in mon_init
void main(void)
{
    //call mon_init()
    //create six producer threads
    //create six consumer threads
}
void producer(void)
{
    while(1) { mon_insert('X'); }
}
void consumer(void)
{
    while(1) { mon_remove(); }
}

//monitor.c
void mon_enter()
{ down(&mutex); }
void mon_exit()
{ up(&mutex); }
void mon_insert(char alpha)
{
    mon_enter();
    if (count == N) wait(full);
    insert_item(alpha); //insert alpha into buf
    count = count + 1;
    if (count == 1) signal(empty);
    mon_exit();
}
void mon_remove()
{
    mon_enter();
    if (count == 0) wait(empty);
    remove_item();  //remove an item from buf
    count = count - 1;
    if (count == N - 1) signal(full);
    mon_exit();
}
```

Figure 2: Pseudo-code for producer/consumer problem using monitors