

## Homework 3: Implementation of Monitors (100 points)

### 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 inserts character 'X' into a circular buffer of size 10,000,000 characters. Each consumer thread removes the last-inserted character from the buffer. Each thread then repeats the process. You will perform your work in *csce.unl.edu*. Use POSIX semaphore (*sem\_init*, *sem\_get*, *sem\_post*, etc.). The pseudo-code is given in Figure 1. Note that the pseudocode only suggests a guideline. Feel free to add more parameters, variables, and functions as they become necessary.

### 2 Producer/Consumer with Monitors (65 pts)

The goal of this problem is to create your own monitor to provide synchronization support for a producer/consumer problem. You will use the pthread package to create 10 producer threads and 10 consumer threads. Each producer thread inserts a randomly generated alphabet (upper and lower cases) into the first available slot a circular buffer of size 10,000,000 characters. Each consumer thread removes a character from the last used slot of the buffer. Each thread then repeats the process. You will perform your work in *csce.unl.edu*. Refer to Figure 2 for the pseudo-code. Note that the pseudocode only suggests a guideline. Feel free to add more parameters, variables, and functions as they become necessary.

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 called *cond*. The structure consists of an integer variable that indicates the number of threads 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 threads blocked on the cv.
  - (b) *wait(cv)*—relinquishes exclusive access to the monitor and then suspends the executing threads.
  - (c) *signal(cv)*—unblocks one thread suspended at the head of the cv blocking queue. The signaled thread **resumes execution where it was last suspended**.

Notice that you **are not allowed** to use existing condition variables such as one from pthread library (*pthread\_cond\_init*). In addition, pay special attention to the following questions during your implementation:

- (a) How would you guarantee that only one thread is inside the monitor at one time?
- (b) Will your monitor follow the *signal and wait* or *signal and continue* discipline?
- (c) How would you make sure that a suspended thread (due to *wait*) resumes where it left off?
- (d) How would you initialize the necessary data structures to support your monitor and make them visible to all threads?

2. You will create function `mon_insert` that inserts a character into the buffer. If the buffer is full, it invokes `wait` on the condition variable *full*. It also invokes `signal` on condition variable *empty*.
3. You will create function `mon_remove` that removes a character from the buffer. If the buffer is empty, it invokes `wait` on the condition variable *empty*. It also invokes `signal` on condition variable *full*. The function returns the removed character.

These functions and your CV 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.
  - Specify whether your monitor follows *signal and wait* or *signal and continue* discipline.
  - How much time you spent on each problem.
  - The level of challenge from 0 to 5 (5 is most difficult) for each problem.
  - How much prior knowledge do you have to attack the problem.
4. Once all solutions are properly stored, zip the folder *lastname\_hw2* and submit the zip file through handin.

```
#define N 10000000
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 to the first available slot in the buffer
        insert('X');
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    while(1)
    {
        down(&full);
        down(&mutex);
        // remove X from the last used slot in the buffer
        remove();
        up(&mutex);
        up(&empty);
    }
}
```

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

```

// ===== pro_con.c ===== //
void main(void)
{
    // any functions you think necessary
    // create ten producer threads
    // create ten consumer threads
}
void producer( ) // add more parameters as needed
{
    char alpha;
    while(1) {
        alpha = generate_random_alphabet();
        mon_insert(alpha);
    }
}
void consumer( ) // add more parameters as needed
{
    char result;
    while(1) { result = mon_remove(); }
}

// ===== monitor.c ===== //
#define N 10000000
cond empty, full;
int count = 0;
char buf[N];
// add more variables as necessary
// (e.g. a semaphore to regulate monitor enter and exit)

void mon_insert(char alpha)
{
    while (count == N) wait(full);
    insert_item(alpha);
    count = count + 1;
    signal(empty);
}
char mon_remove()
{
    char result;
    while (count == 0) wait(empty);
    result = remove_item();
    count = count - 1;
    signal(full);
    return result;
}

```

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