CSCE 451/851 Programming Assignment 3

Implementation of a Monitor

1 Overview of Project

This programming assignment is to familiarize yourself with the synchronization constructs we discuss in class – mutexes, semaphores, and monitors. The goal is to ensure no deadlocks, no race conditions, and that data is shared properly between threads.

Part 1 In Part 1 you will solve the producer/consumer problem by utilizing POSIX semaphores to solve concurrency problems.

Part 2 In Part 2 you will solve the producer/consumer problem by developing a custom monitor.

As usual we have provided you with precompiled binaries that run on the CSE servers to observe the correct input/output and compare against your solution.

2 Program Specification

Implement switches To make the program easier to evaluate, you will need to implement additional switches to specify the buffer length, number of producers and the number of consumers as indicated below.

Switch	Specifies
-b	Buffer length in bytes
-p	Number of producer threads
-c	Number of consumer threads
-i	Number of items to insert

As an example: `cse> ./part1 -b 1000 -p 10 -c 10 -i 100000`
should create a buffer of length 1,000 bytes, 10 producer threads, 10 consumer threads, and insert 10,000 items.

Output Convention Whenever an item is produced (i.e., put in the buffer), a message needs to be printed to the screen with the following (`printf()`) convention:

```
"p:<%u>, item: %c, at %d", threadid, item, index
```

This indicates a producer with `threadid` inserted `item` at `index` in the buffer. A similar message is required whenever an item is produced:

```
"c:<%u>, item: %c, at %d", threadid, item, index
```

indicating a consumer with `threadid` has consumed `item` at `index` in the buffer.

Exit The producer thread should exit when it has no items left to insert into the buffer. The consumer thread should exit when the buffer is empty and it has consumed all the items. Manage this by using a counter for the number of items inserted and removed.

3 Submission

Use CSE web handin to hand in your assignment. Submit a single zip file, `<UNL username>_<pa#>.zip` (e.g., `jdoe2_pa1.zip`) containing the following directory structure:

```
<UNL username>_pa3
|-----prob1
|      |----Makefile
|      |----part1.c
|-----prob2
       |----Makefile
       |----part2.c
```

Be sure to:

1. Replace `<UNL username>` with your UNL username.
2. Add a separate `Makefile` for both problems. The `Makefile`s should:
   a. Have an `all` target as the first target.
   b. Produce respective binaries `part1` for Part 1, and `part2` for Part 2.

4 Evaluation

Your program will be graded according to the following rubric:

Command	ND?	NRC?	ICM?	Total
`./part1 -b 1 -p 5 -c 5 -i 10`	5	5	5	15
`./part1 -b 1000 -p 20 -c 20 -i 10000`	5	5	5	15
Part 1 Total = 30 points				
`./part2 -b 1 -p 5 -c 5 -i 10`	5	5	5	15
`./part2 -b 4 -p 30 -c 30 -i 10`	5	5	5	15
`./part2 -b 10 -p 1 -c 10 -i 20`	5	5	5	15
`./part2 -b 1000 -p 20 -c 20 -i 10000`	5	5	5	15
Part 2 Total = 60 points				
`Makefile` for Part 1				5
`Makefile` for Part 2				5
Total				**100**

where

- "ND?" = No Deadlocks?
  - Deadlocks would cause your program to hang, so we will look for that.
- "NRC?" = No Race Conditions?
  - Race conditions would manifest most typically by producing a character in the buffer at a certain index and then consuming from that same index a different character. So be sure if a producer puts a 'D' at index 5 in the buffer, a consumer gets a 'D' out at index 5.
- "ICM?" = Item Count Met?
  - You should consume and produce the same number of items, which should match the command line argument for `-i`.

Additionally if the naming convention described in the Submission section above is not followed you lose 10 points. Remember, if your code doesn't compile on the CSE servers you will get 0 for all tests. ***Please*** check that your code both compiles and runs on the CSE servers!!! Don't wait until it works completely on your own machine before testing on the CSE servers. Test on the CSE servers at regular milestones in your development.

Protip You can combine the `grep` command with the `wc` command to get the count of producers or consumers. For example, for producers:

```
./part2 -b 1000 -p 20 -c 20 -i 10000 | grep p: | wc -l
```

And for consumers:

```
./part2 -b 1000 -p 20 -c 20 -i 10000 | grep c: | wc -l
```

5 Part 1: Producer/Consumer with Semaphores

The goal of Part 1 is to solve the producer/consumer problem using semaphores. You will use the pthread library to create producer threads and consumer threads. Each producer thread inserts a single 'X' character into a buffer. Each consumer thread removes the most recently inserted 'X' from the buffer. Each thread then repeats the process. Use POSIX semaphore (`sem_init`, `sem_wait`, `sem_post`, `sem_destroy`). The pseudocode is given in Pseudocode 1 below. *Please stick to the main structure in the pseudocode, but feel free to add more parameters, variables, and functions as they become necessary.*

Pseudocode 1

Pseudocode for producer/consumer problem using semaphores:

```
# define N 10000000
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void main( void )
{
    // create -p # of producer threads
    // create -c # of consumer threads
}

void * producer( void )
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&mutex);
        // insert X into the first available slot in the buffer
        insert('X');
        sem_post(&mutex);
        sem_post(&full);
    }
}

void * consumer( void )
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&mutex);
        // remove X from the last used slot in the buffer
        remove();
```

```
        sem_post(&mutex);
        sem_post(&empty);
    }
}
```

6 Part 2: Producer/Consumer with Monitor

The goal of Part 2 is to create your own monitor to provide synchronization support for the producer/consumer problem. You will use the pthread library again to create producer threads and consumer threads. Each producer thread inserts a randomly generated character from the alphabet (upper and lower cases) into the first available slot in a buffer. Each consumer thread removes a character from the most recently used slot of the buffer. Each thread then repeats the process. *Pseudocode 2 provides the basic outline your code should take, but feel free to add more parameters, variables, and functions as they become necessary.*

Pseudocode 2

Pseudocode for producer/consumer problem using monitors:

```
// ===== pro_con.c ===== //
void main( void )
{
    // any functions you think necessary
    // create producer threads
    // create consumer threads
    // join all 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

typedef struct {
    // condition variable fields
} cond;

int count(cond cv)
{
    // return # of threads blocked on cv
}

void wait(cond cv)
```

```
{
    // give up exclusive access to monitor
    //      and suspend appropriate thread
    // implement either Hoare or Mesa paradigm
}

void signal(cond cv)
{
    // unblock suspended thread at head of queue
    // implement either Hoare or Mesa paradigm
}

cond empty, full;
int count = 0;
char buf[N];
// add more variables as necessary
// define condition variable struct
// define monitor struct

void mon_insert(char alpha)
{
    // implement either Hoare or Mesa paradigm

    // synchronization and bookkeeping
    while(count == N)
        wait(full);
    insert_item(alpha);
    count = count+1;
    signal(empty);
}
char mon_remove()
{
    // implement either Hoare or Mesa paradigm

    // synchronization and bookkeeping
    char result;
    while(count == 0)
        wait(empty);
    result = remove_item();
    count = count-1;
    signal(full);
    return result;
}
```

Helpful Steps

1. Create a new variable type for condition variables (CV). CVs are used to delay processes or threads that cannot continue executing due to a specific monitor state (e.g., full buffer). They are also used to awaken delayed processes or threads when the conditions are satisfiable. The variable type (probably a `struct`) 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:
    - `count(cv)`—returns the number of threads blocked on the `cv`.
    - `wait(cv)`— relinquishes exclusive access to the monitor and then suspends the executing

threads.
- `signal(cv)`—unblocks one thread suspended at the head of the cv blocking queue. The signaled thread resumes execution where it was last suspended.

You **are not allowed** to use existing condition variables or synchronization mechanisms such as the one from pthread library (`pthread_cond_init`, or any of the other `pthread_` functions) You should be able to complete the assignment with:

- `sem_wait()` - decrement (lock) a semaphore
- `sem_post()` - increment (unlock) a sempahore
- `sem_init()` - initialize a semaphore
- `sem_destroy()` - destroy the semaphore
- `pthread_create()` - create a thread
- `pthread_exit()` - exit a thread

If you use other `pthread_` function calls you will lose points.
Think about the following questions as you design your solution (they make good test questions):

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. Create a 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. Create a 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.