

CSCE 451/851 Programming Assignment 3

Implementation of a Monitor

Assigned: Feb 20, 2020

Due: Mar 19, 2020 23:59:59

Introduction

This programming assignment is to get you familiar with the synchronization constructs we've been discussing in class – mutexes, semaphores, and monitors. There are two problems in this assignment: solving the producer/consumer problem with 1) semaphores, and 2) your own custom monitor.

Submission procedure

Do the following:

1. Create a directory called `<UNL username>_pa3` (replace `<UNL username>` with your UNL username).
2. Create subdirectories: `prob1` and `prob2` under `<UNL username>_pa3`.
3. Place your solutions in the proper directory. Provide a `README` file for each problem in each directory. Each `README` file should specify:
 - a. Instructions on how to test your solution.
 - b. Whether your monitor follows the “signal and wait” or the “signal and continue” paradigm and why you chose it.
4. Add a separate `Makefile` for both problems. The name of the executable produced by the first part and second part should be `part1` and `part2` respectively. **Note that if you don't provide a `Makefile` we can't/won't compile your code and you won't pass any tests.**
5. You should have the following directory structure:

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

6. Zip the directory `<UNL username>_pa3` and submit.

Use web handin to hand in your assignment. Submit a single zip file, `<UNL username>_pa3.zip` (e.g., `jdoe2_pa3.zip`).

Program Specification

Implement switch 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 inserted or removed from the buffer, a message needs to be printed to the screen with the following convention:

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

This is to indicate the producer thread with `threadid` inserted an item at the index in buffer. We will output a similar message for the consumer when an item is consumed, replacing "p" with "c".

Exit The producer thread should exit when it has no items to be inserted to 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.

Provided Binaries 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.

Evaluation

Your program will be graded according to the following rubric:

Command	ND?	NRC?	ICM?	Total
<code>./part1 -b 1 -p 5 -c 5 -i 10</code>	5	5	5	15
<code>./part1 -b 1000 -p 20 -c 20 -i 10000</code>	5	5	5	15
Part 1 README				5
Part 1 Total = 35 points				
<code>./part2 -b 1 -p 5 -c 5 -i 10</code>	5	5	5	15
<code>./part2 -b 4 -p 30 -c 30 -i 10</code>	5	5	5	15
<code>./part2 -b 10 -p 1 -c 10 -i 20</code>	5	5	5	15
<code>./part2 -b 1000 -p 20 -c 20 -i 10000</code>	5	5	5	15
Part 2 README				5
Part 2 Total = 65 points				

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. If your code does not compile on the CSE servers you can only get the points for the README, which is 10 max. In other words, 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.

Finally, you can combine the `grep` command with the `wc` command to get the count of producers or con-

sumers. 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
```

1 Producer/Consumer with Semaphores (35 pts)

The goal of this problem 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 100000000
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)
    {
        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);
    }
}

```

2 Producer/Consumer with Monitor (65 pts)

The goal of this problem 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
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)
{
    // synchronization and bookkeeping
    while(count == N)
        wait(full);
}

```

```

        insert_item(alpha);
        count = count+1;
        signal(empty);
    }
    char mon_remove()
    {
        // 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 called for condition variables (CV). Condition variables 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 (i.e., 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 semaphore
- `sem_init()` - initialize a semaphore
- `sem_destroy()` - destroy the semaphore

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.