Introduction to Multithreading - Part 02

M.D.R.A.M De Silva
E/13/058
Semester 06

Exercise 01

1. What is the final result for myglobal in the above code? If the threads ran one after the other, what would be the final result for myglobal ? What would happen if the sleep() statement in the thread_function() is removed ? Explain these differences.

```
rangana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$ gcc -Wall -pthread ra
ce.c -o race
rangana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$ ./race
..................................................
myglobal equals 5
rangana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$
```

Final result of my global is 5.
If the threads ran one after the other than my global will be 50.

When sleep() statement in thread function is removed,

```
rangana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$ gcc -Wall -pthread r
ce.c -o race
rangana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$ ./race
..................................................
myglobal equals 42
rangana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$
```

My global is in this output is 42. It can be a value between 5- 50.

When there is a sleep () condition inside the thread function ,at first ten threads are created and they are run separately and each of these threads increment the counter value by one. And since we make sure that every thread write the same value to the counter they'll be no errors. Hence, It will return 5 as all the threads write the same.

Unlike this, if threads run one after the other the counter will increment by 5 thread after thread making the final value of myglobal variable 50.

When we remove the sleep() statement in the thread function there is no guarantee that the threads will run one after the other and there will also be a race condition to increment the value of myglobal . . Not all threads will write the same value. Because of this condition the value of my global will not update as required and it will be a value between 5-50.The value obtained for

the variable myglobal in this case depends on how the OS allocate time for different threads to update the variable.

2. If we modified the code,Will that resolve the problem? Explain why.
After modifying,

```
rangana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$ gcc -Wall -pthread ra
ce.c -o race
rangana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$ ./race
..................................................
myglobal equals 50
rangana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$
```

By doing this modification we remove the aspect of incrementing the variable separately and each time thread access global variable it will increment the value.This will result a value near to 50 and less than 50 when there is a race condition.If the scheduler handles the process without any race condition the final value will be 50.

3. Can you write the same program using multiprocessors (without using interprocess communication methods)? Explain why.

No.
Without the use of inter process communication it is not possible write the same program as for each process created there will be a new variable created and that new variable will be the one incremented each time. There wouldn't be a shared variable .And that will not provide the required output of the given program.

Exercise 02

Norace.c before modification,

```
rangana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$ gcc -Wall -pthread no
race.c -o norace
rangana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$ ./norace
..................................................
myglobal equals 50
rangana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$
```

After modifying,

```
angana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$ gcc -Wall -pthread no
ace.c -o noracechange
angana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$ ./noracechange
..................................................
yglobal equals 5
angana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$
```

Reason :

The result is not the same. We see that after modification myglobal value has become 5 and not 50.

```
void *thread_function(void *arg) {
    int i,j;
    for ( i = 0; i < 5; i++ ) {
        pthread_mutex_lock(&mutex);
        j = myglobal;
        pthread_mutex_unlock(&mutex);
        j = j+1;
        printf(".");
        fflush(stdout);
        sleep(1);
        pthread_mutex_lock(&mutex);
        myglobal = j;
        pthread_mutex_unlock(&mutex);
    }
```

Here,once the thread function is called by a created thread it will get the mutex lock and then assign the global variable to j and release the lock which lets other threads to use the lock. Then what happens is other threads gets the opportunity to access the global variable.Since the lock is released by now all the 10 threads gets to execute up to sleep(1) command .Then each thread acquires the lock separately and update the global variable(increment it by one) and release it . All the ten thread do this. All the ten threads after finishing thread function will only increment the global variable by 5 and hence the output is 5.

Exercises 03

1. Remove the mutex from the above code. Run it and see what happens.

Before removing,

```
rangana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$ gcc -Wall -pthread bo
undedbuffer_producerconsumer.c -o bounded
rangana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$ ./bounded
Producer produced 1
Consumer consumed 1
Producer produced 1
Consumer consumed 1
Producer produced 1
Consumer consumed 1
Producer produced 1
Consumer consumed 1
Producer produced 1
Consumer consumed 1
Producer produced 1
Consumer consumed 1
Producer produced 1
^C
rangana@rangana-VirtualBox:~/Documents/co327/lab 04/lab04$
```

Let me explain what happens here. When the main thread is executed it will first create three semaphores,
1.item --> value = 0
2.spaces → value = 20
3.lock → value = 1

And then it will invoke the thread function,consumer.

```
void* consumer(void *arg) {
          while(1) {
            sem_wait(&items);
            sem_wait(&lock);
            printf("Consumer consumed %d\n",products); // work with the resource
            sleep(1);
            products--;
            sem_post(&lock);
            sem_post(&spaces);
            sleep(1);
          }
          return NULL;
```

When sem_wait(&item) is called ,since item value is 0 at the beginning it will make it -1 and will wait till some items get produced. Since it is waiting it will not enter the critical section of the consumer.

Then the producer's role come into play.

```
while(1) {
  sem_wait(&spaces);
  sem_wait(&lock);
  products++;
  printf("Producer produced %d\n",products);
  sleep(1);
  sem_post(&lock);
  sem_post(&items);
  sleep(1);
}
```

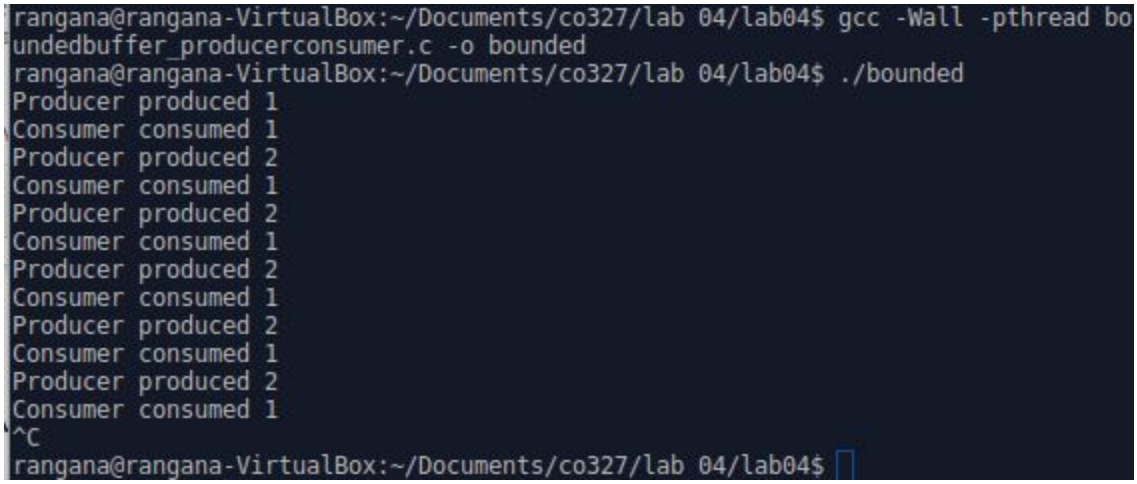
When sem_wait(&spaces) is called it will decrease the amount of spaces by one. Then sem_wait(&lock) is called. Lock will become zero.Then a item is produced (product ++) and the first print statement ("producer produced 1") is shown in console.

Then, sem_post(&lock) is called and the lock value is set to 1 and by sem_post(&items) item value is set to 0 allowing consumer to continue with the thread function.

Then the consumer executes sem_wait(&lock) to make it to 0.and consumes an item and print "Consumer consumed 1".
Then by sem_post(&lock) the lock value is set to 1 and by sem_post(&spaces) spaces is incremented by one.

This procedure goes on and thats why the console output looks like that.

After,



This can be shown using the following tables(code structure without the mutex.)

Producer

p(spaces)
product++
printf
v(item)

Consumer

p(item)
printf
product--
v(spaces)

At the beginning ,
Items = 0
Spaces = 20
No mutex.Hence no lock variable.

Consumer tries to consume item and call p(item) which makes item = -1 and consumer waits without going to the critical section.

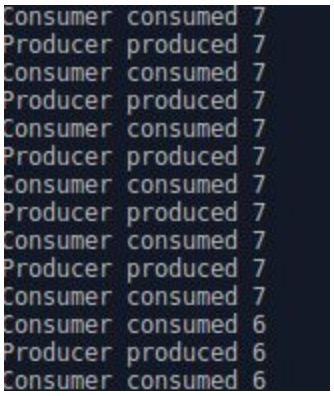
Then a producer tries to produce an item.
First it calls p(spaces)which makes spaces = 20-1 = 19.
Then it produce an item and print(“ Producer produced 1”).
And call v(item) which makes item = 0 and allows consumer to continue.

Then the consumer print (consumer consumed 1) and reduce the product count by one.
Then by calling v(spaces) it increase available spaces. Here, reducing the product count is the critical section and what happens is , before this happens due to not properly synchronising producer will produce another one making products = 2 and print (producer produced 2).And then only wi;; consumer will reduce one item. And then it will increase the spaces and run p(item) and print (consumer consumed 1) and by no products= 1 and at this point again the producer

increase the product count and print (producer produced 2) This will repeat as seen in the console output.We see that without the lock product variable is free to access by anyone.

In order to avoid this we need to declare a lock mutex variable to allow only to access the critical section at a time and to act correctly when there is a context switch. This can be done by a lock as ,producer will not be able to increase product count when the consumer keeps the lock . This will work even when there is a context switch.

2. Compile and run the modified version of the above code (producerconsumer.c in the tarball). Is there any difference in the output? Explain the results.



Output is not the same it is different.The sleep() time is set to a random value in the code and hence , semaphore waits and posts in different amounts of time. Because of this,sometimes producer gets the chance to execute more times before the consumer gets a chance and vise versa. And that explains the output of this program.

3. Is there any specific reason for the order of semaphores signaling above? Switch the order of semaphores and see what happens. Explain.
Lock semaphore is the key.It should be placed after checking for empty items and full spaces in order to avoid deadlocks as in the code above.

Lets check this scenario by changing the order,
Assume consumer starts proceedings.

Producer

p(lock)
p(spaces)
Produce item(product++)
v(item)
v(lock)

Consumer

P(lock)
P(item)
Consume item(product--)
v(spaces)
v(lock)

At the beginning,

Spaces = 20
Item = 0
Lock = 1

Consumer runs p(lock),which makes lock = 0
Consumer runs p(item),which makes item = -1, consumer waits ….

Producer runs p(lock) ,which makes lock = -1, producer waits…
Both processes waiting. This will lead to a dead lock.

This will work if the producer starts the proceeding and it will fail when consumer starts the proceedings . Hence the order is critical and we should make sure that follow this order,(which is also the same order in the code)

Producer

p(spaces)
p(lock)
Produce item(product++)
v(lock)
v(item)

Consumer

P(lock)
P(item)
Consume item(product--)
v(lock)
v(spaces)

Exercise 04

1. Implement above mentioned reader-writer situation using semaphores. Assume that, for this question the data resource is an array. A reader would read a random integer from it and prints its value to the screen while a writer increments a value at an arbitrary index and prints the value.

In the submission folder

The main drawback of this implementation of the reader-writer problem is that it can lead to writer starvation. One solution to overcome this is to have an implementation with “writer-preference”. However, such a solution would introduce reader-starvation.

2. Discuss possible solutions to avoid starvation for any reader or writer, comparing their relative merits. Implement one of those solutions.