CO327 LAB04

KANEWALA U.C.H.

E/13/175

SEMESTER 6

08/09/2017

INTRODUCTION TO MULTITHREADING – PART II

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.**

Final result of above code : myglobal = 5
Threads running one after another: myglobal = 50
sleep() statement removed: myglobal = 24 (a value between 5 – 50)

In the original code, there is no delay in creating the 10 threads. This makes each thread get the same initial value for the myglobal variable and thus for the variable j. But, there is a 1ms sleep time included in the thread_function() before writing the value to the variable after incrementing separately. This makes all the threads get the same value for myglobal after each increment.Hence, after incrementing all the threads will be writing the same value to the variable myglobal and there will be no errors in its value due to this reason. Hence, the final result will be 5 for variable myglobal.

When the threads run after another, we can see a sequential increment in the myglobal variable. Each thread will increment the value of myglobal and print a dot before the execution of the next thread. Hence, myglobal will be 50.

When the sleep() statement is removed, the threads running concurrently will update the value of myglobal variable. This will give rise to race conditions. Multiple threads will try to write the same value or different values depending on the scheduling. Hence, the variable myglobal will show an erroneous value.

2. **If we modified the following code:**

```
j = myglobal;
j = j + 1;
printf(".");
fflush(stdout);
sleep(1);
myglobal = j;
```

in to

```
myglobal = myglobal +1;
printf(".");
fflush(stdout);
sleep(1);
```

Will that resolve the problem? Explain why.

Yes. In this case, the value of the variable my global is updated as soon as a thread executes the thread function. In Earlier code the value of j is incremented as soon as the thread function is executed but the myglobal variable is updated after 1 ms sleep time. This will make the all 10 threads receive an initial value of myglobal as 0. Hence each thread will increment myglobal variable from 0 to 5 sequentially as everytime all threads get the same myglobal value when considering parallel iterations.

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

No. A shared space cannot be handled without inter process communication. Hence, without a shared space, a new global variable will be created each time when a new process I created using threads. Hence, each will be incrementing a separate variable. Therefore it is not possible to write the same program without inter process communication.

Exercise 02

if we used:

```
pthread_mutex_lock(&mymutex);
j = myglobal;
pthread_mutex_unlock(&mymutex);
j = j + 1;
printf(".");
fflush(stdout);
sleep(1);
pthread_mutex_lock(&mymutex);
myglobal = j;
pthread_mutex_unlock(&mymutex);
```

instead of:

```
pthread_mutex_lock(&mymutex);
j = myglobal;
j = j + 1;
printf(".");
fflush(stdout);
sleep(1);
myglobal = j;
pthread_mutex_unlock(&mymutex);
```

Will that give you the same result? Explain.

No. The result is myglobal = 5. Earlier case myglobal = 50.

This is due to the sleep time included in the thread_function(). Since the value of my global is updated after the sleep time, all 10 threads will get the same initial value which is equal to zero during this sleep time. The same behaviour is continued in next iterations as well. Hence the global value will be 5.

Exercises 03

1. Remove the mutex from the above code. Run it and see what happens. Why is it needed?

The products start showing incorrect value after the first consumer consumes 1 product. The error continues. This is because without the locked mutex the products variable can be accessed by anyone freely. In this case, it shows erroneous values as the main thread with producer function increments the products variable before the thread_function decrements the products variable in the previous consumption due to the context switches.

It is needed to provide mutual exclusion to the shared variable products in order to get the correct functionality of the producer and consumer. The mutex lock makes sure only one thread is running inside the critical section where it is placed to lock. Here, it makes sure the products variable is updated only by one either the producer or consumer at a given time.

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.

Here, we can see the producer and consumer operating randomly as the sleep time giving the context switch is random. Also, we can see that the products variable increments after a random time and reach up to full capacity 20 and reducing back again to 0. The increment occurs when two consumers produce one after the other. The decrement occurs when two consumers consume one after another. When a producer and consumer are switching turns one by one we see the same value for the product the entire time. In the earlier code, this producer consumer switching one after another was seen as there are only 1 producer and 1 consumer with equal sleep times

3. Is there any specific reason for the order of semaphores signalling above? Switch the order of semaphores and see what happens. Explain.

Yes. the spaces semaphore is a counting semaphore indicating the capacity of the bounded buffer. It prevents the producer producing until there is available space in the buffer. In the main function, the producer waits until there are available spaces keep the produced items. The producer continues production by incrementing the products variable the main function. In the thread_function() the items semaphore is a binary semaphore indicating whether there are items produced or not. Therefore, in the given order the consumer waits if there are no items produced. The producer signals the waited threads after producing each time indicating that there are available items for consumption. Switching the items and spaces semaphore will result in giving deadlocks.

The lock semaphore is providing mutual exclusion needed for the product variable. Switching this with any other semaphore might result in deadlocks at certain cases. Ideally, the lock should be placed after checking for empty spaces and after checking for available products for consumption. Switching this order can result in a dead lock if the products variable is locked when the there are no items to be consumed or when there is no space to produce. Then the producer will be waiting trying to get access forever in the first case and the same will be for the consumer in the second case respectively.

When unlocking also it is better to unlock the semaphore providing mutual exclusion lock in this case first. This is because, if the two synchronizing semaphores are unlocked first and a context switch occurs, the critical section will be locked until the context switch happens back

to the same thread keeping the lock. This can lead to dead locks in certain cases. These are rare situations but if occurred can lead to a dead lock.

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.**

   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.**

   In the reader-writer problem, depending on the requirement of the application the priority can be given to readers or writers. In either case, the party with lesser priority will face starvation while the privileged party gets instant access. This can be minimized by providing the reader and writer equal opportunity to gain access by placing them in a queue. This queue is only a gateway to gain access to the shared data. This idea of a queue does not affect the readers who are reading simultaneously. It only affects for the attempt taken to access the data at first. That is once a reader or writer gains access to the data waiting in the queue, the respective reader or writer is removed from waiting. In this way, once a writer comes to the queue, the writer will have to wait until all readers who have previously gained access exits from the shared data and until that no other new readers are allowed to access the shared data as well. Also, once a writer gets to access the readers coming after will not be able to access shared data until the writer finishes writing and exciting. This solution is implemented in reader_writer_withoutStarvation.c file.

   In the above solution also there is a starvation factor included for the writer when the writer is about to access the shared data or for readers when the readers are trying to access when a writer is already writing to the shared data. This can be avoided by including a buffer to write the new data so that writers can write to the buffer while the previous readers read from the original old data. Then, after all the previous reader's exit, the original can be replaced with the buffer content or the previous can be deleted if the new buffer can be pointed as the original. In this case, until the new content is written to the buffer all readers coming will be directed to the old copy whereas once the writing is finished, the readers coming after will be directed to the new copy. In this solution, additional space is required for each write operation and for a critical application which needs to read the latest updated data this method will not be suitable.