Introduction to Multithreading – Part II

DEPARTMENT OF COMPUTER ENGINEERING

Race conditions and Synchronization

A race condition, as applied to a software system, is the behavior of the system such that the output is dependent on the sequence of timing or other uncontrollable events. This happens when more than one independent sequence of executions use the same resource. In our application, we will consider race conditions where the shared resource is the program memory.

To resolve race conditions, the independent sequences of commands should be synchronized. i.e., it should be coordinated such that, differences in timing does not affect the final result of the system.

Let us consider the following example to understand how a race condition works.

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int myglobal = 0;

void *thread_function(void *arg) {
       int i, j;
       for (i = 0; i < 5; i++ ){
              j = myglobal;
              j = j + 1;
              printf(".");
              fflush(stdout);
              sleep(1);
              myglobal = j;
       }
       return NULL;
}

int main(void) {
       pthread_t mythreads[10];
       int i;
       for(i = 0; i < 10; i++){
              if ( pthread_create(&mythreads[i], NULL, thread_function, NULL) ) {
                     printf("error creating threads.");
                     abort();
              }
       }
       for(i = 0; i < 10; i++)    {
              if ( pthread_join ( mythreads[i], NULL ) ) {
                     printf("error joining threads.");
                     abort();
              }
       }
       printf("\nmyglobal equals %d\n",myglobal);
       exit(0);
}
```

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.
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.

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

Thread synchronization

Thread synchronization is the process of applying certain mechanisms such that the threads access shared resources in an order that does not cause a race condition. There are few such mechanisms available in the `pthread` library. Such as:

1. Thread joining
2. Mutexes
3. Semaphores
4. condition variables

From the above three, we have already discussed thread joining. In this lab session we will discuss mutexes and semaphores.

1. mutex

A mutex ensures that only a single thread could access a resource at a time. If more than the allowed number of threads try to access the resource, those threads are sent to sleep. When one thread finishes using the resource, one of the sleeping threads is woken up and is allowed to access the resource.

Let us rewrite the above code corrected with a *mutex*.

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int myglobal;
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

void *thread_function(void *arg) {
       int i,j;
       for ( i = 0; I < 5; i++ ) {
              pthread_mutex_lock(&mutex);
              j = myglobal;
              j = j + 1;
              printf(".");
              fflush(stdout);
              sleep(1);
              myglobal = j;
              pthread_mutex_unlock(&mutex);
       }
       return NULL;
}
int main() {
       pthread_t mythreads[10];
       int i;
       for(i = 0; i < 10; i++){
              if ( pthread_create(&mythreads[i], NULL, thread_function, NULL) ) {
                     printf("error creating threads.");
                     abort();
              }
       }
       for(i = 0; i < 10; i++)    {
              if ( pthread_join ( mythreads[i], NULL ) ) {
                     printf("error joining threads.");
                     abort();
              }
```

```
        }
        printf("\nmyglobal equals %d\n",myglobal);
        exit(0);
}
```

`pthread_mutex_t` is the struct that defines a POSIX mutex.

`PTHREAD_MUTEX_INITIALIZER` macro

The `PTHREAD_MUTEX_INITIALIZER` macro initializes a static mutex, setting its attributes to default values. `pthread_mutex_init(pthread_mutex_t*, const pthread_mutexattr_t*)` can also be used to initialize a mutex. Setting the second argument to NULL will set the default values as attributes.

`pthread_mutex_lock()` call

The mutex object referenced by `mutex` is locked by calling `pthread_mutex_lock()`.

If the mutex is already locked, the calling thread blocks until the mutex becomes available. That is, the calling thread waits until the thread locking the variable unlocks it, preventing a possible race condition. Attempting to relock the mutex causes a deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, the resulting behavior is undefined.

`pthread_mutex_unlock()` call

Unlocks a locked mutex. i.e. allows other threads to lock the mutex which indicates the other thread that the shared variable is safe to modify. If there is a thread that has called `pthread_mutex_lock()` and sent to sleep, this call will wake it up.

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.

Semaphores

Semaphore is a similar mechanism to the mutex but could handle more than one resource at a time. It contains a `v()` (also called `wait()`) and `p()` (also called `signal()` or `post()`) functions which are used for waiting for a resource and signaling a waiting process that a resource is available respectively.

`pthread` library does not implement any semaphore mechanism but POSIX semaphores are available to use.

Let us consider a classic example of synchronization problem that could be easily solved by using semaphores.

Producer-consumer problem
The problem describes two processes, the producer and the consumer, who share a common, buffer. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time. The problem is to make sure that the consumer won't try to remove data from an empty buffer and to make sure that all the values consumed are valid.

Consider the following code:

```
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

sem_t items;

void* consumer(void *arg) {
	while(1) {
		sem_wait(&items);
		printf("Consumer consumed\n");
		sleep(1);
	}
	return NULL;
}

int main(int argc, char **argv) {
	pthread_t thread;
	if(sem_init(&items, 0, 0)){
		printf("Could not initialize a semaphore\n");
		return -1;
	}
	if(pthread_create(&thread, NULL, &consumer, NULL)) {
		printf("Could not create thread\n");
		return -1;
	}
	while(1) {
		printf("Producer produced\n");
		sleep(1);
		sem_post(&items);
	}
	return 0;
}
```

The `sem_init()` call
This function initializes the semaphore. The first argument for this function is a pointer a semaphore variable while the second one chooses whether we're using it for synchronization of few processes or few threads that are inside the same process. We will not discuss the former. The third argument specifies the initial value of the counter variable.

`sem_wait()` call
The `wait()` functionality described above

`sem_post()`
the `signal()` functionality described above

`sem_destroy()` call
Deletes a semaphore

In the above code, you might notice that the main thread keeps signaling semaphore infinitely after printing. The printing simulates the producing. i.e., here the main thread is modeled as the producer. The signaling of semaphore tells a consumer that the producer has produced. Since this increments the semaphore, its count works as the variable that keeps track of the number of items produced. Since the main thread keeps signaling semaphore infinitely, this could be considered as a model code for unbounded buffer producer and consumer problem. The shared resource here is the semaphore itself.

Bounded buffer producer and consumer problem with semaphores

Let's consider the bounded buffer producer-consumer implementation with semaphores. Here, not only whether there are any items to be consumed should be kept track of, but since the space for a produced item is also limited, the number of spaces should also be considered. Therefore, we define another semaphore to track the number of spaces.

Consider the following code:

```
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#define CAPACITY 20
sem_t items;
sem_t spaces;
sem_t lock;
int products = 0;

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;
}

int main(int argc, char **argv) {
            pthread_t thread;
            if(sem_init(&items, 0, 0)) {
              printf("Could not initialize a semaphore\n");
              return -1;
            }
            if(sem_init(&spaces, 0, CAPACITY)) {
              printf("Could not initialize a semaphore\n");
              return -1;
            }
            if(sem_init(&lock, 0, 1)) {
              printf("Could not initialize a semaphore\n");
              return -1;
            }
            if(pthread_create(&thread, NULL, &consumer, NULL)) {
              printf("Could not create thread\n");
              return -1;
            }
            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);
            }
            return 0;
}
```

Note that, in above code, we have introduced a variable called *products* to represent the items instead of treating the semaphore itself as the items. Therefore, another mutex variable, *lock*, should be introduced to protect this variable.

Exercises 03

1. Remove the mutex from the above code. Run it and see what happens. Why is it needed?
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.
3. Is there any specific reason for the order of semaphores signaling above? Switch the order of semaphores and see what happens. Explain.

The reader-writer problem

The same data resource is read by certain threads (called readers) and written by other threads (called writers) concurrently. To avoid readers reading wrong data and writers corrupting the data resource, only one writer can write at the same time and no readers could read data when a writer is writing. But when there are only readers reading data, any number of readers could read the data concurrently.

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.