Tutorial 5

Operating System Lab

Semaphore

- Semaphore is a simply a variable.
- This variable is used to solve critical section problem and to achieve process synchronization in the multi processing environment.
- The two most common kinds of semaphores are counting semaphores and binary semaphores.
- Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1. only.

Thread Synchronization Problems

- Lets take an example code to study synchronization problems :

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

pthread_t tid[2];
int counter;

void* doSomeThing(void *arg)
{
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d started\n", counter);
```

```
for(i=0;i<1000000;i++);
    printf("\n Job %d finished\n", counter);
    return NULL;
}

int main(void)
{
    int i = 0;
    int err;
    while(i < 2)
    {
        err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
        if (err != 0)
            printf("\ncan't create thread :[%s]", strerror(err));
        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    return 0;
}
```

- The code and the flow looks fine but when we see the output :

```
$ ./testthreads
Job 1 started
Job 2 started
Job 2 finished
Job 2 finished
```

- If you focus on the last two logs, you will see that the log ‘Job 2 finished’ is repeated twice while no log for ‘Job 1 finished’ is seen.

But if you’ll have a closer look and visualize the execution of the code, you’ll find that :

- The log ‘Job 2 started’ is printed just after ‘Job 1 Started’ so it can easily be concluded that while thread 1 was processing the scheduler scheduled the thread 2.
- If the above assumption was true then the value of the ‘counter’ variable got incremented again before job 1 got finished.
- So, when Job 1 actually got finished, then the wrong value of counter produced the log ‘Job 2 finished’ followed by the ‘Job 2 finished’ for the actual job 2 or vice versa as it is dependent on scheduler.
- So we see that it’s not the repetitive log but the wrong value of the ‘counter’ variable that is the problem.

The actual problem was the usage of the variable ‘counter’ by second thread when the first thread was using or about to use it. In other words we can say that lack of synchronization between the threads while using the shared resource ‘counter’ caused the problems or in one word we can say that this problem happened due to ‘Synchronization problem’ between two threads.

Mutex

- A Mutex is a lock that we set before using a shared resource and release after using it. When the lock is set, no other thread can access the locked region of code.
- We see that even if thread 2 is scheduled while thread 1 was not done accessing the shared resource and the code is locked by thread 1 using mutexes then thread 2 cannot even access that region of code. So this ensures a synchronized access of shared resources in the code.

Mutex

Internally it works as follows :

- Suppose one thread has locked a region of code using mutex and is executing that piece of code.
- Now if scheduler decides to do a context switch, then all the other threads which are ready to execute the same region are unblocked.
- Only one of all the threads would make it to the execution but if this thread tries to execute the same region of code that is already locked then it will again go to sleep.
- Context switch will take place again and again but no thread would be able to execute the locked region of code until the mutex lock over it is released.
- Mutex lock will only be released by the thread who locked it.
- So this ensures that once a thread has locked a piece of code then no other thread can execute the same region until it is unlocked by the thread who locked it.
- Hence, this system ensures synchronization among the threads while working on shared resources.

Mutex

A mutex is initialized and then a lock is achieved by calling the following two functions :

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

The first function initializes a mutex and through second function any critical region in the code can be locked.

The mutex can be unlocked and destroyed by calling following functions :

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

The first function above releases the lock and the second function destroys the lock so that it cannot be used anywhere in future.

Mutex vs Semaphore

Using Mutex:

A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa. At any point of time, only one thread can work with the *entire* buffer. The concept can be generalized using semaphore.

Using Semaphore:

A semaphore is a generalized mutex. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.