CHAPTER 9
Process Synchronization

Learning Objectives

1. Students understand the concepts of race condition, critical section, and process synchronization.
2. Students able to implement process synchronization.

Introduction

In a multi-programming system, resources are dynamically shared among a number of active processes. These processes are interleaved and overlapped in time during their execution. Operating systems must optimally manage the allocation and sharing of resources between processes. One of the issues related to resource management is that the relative speed of execution of processes cannot be predicted. Some processes or threads may want to perform read and write operations on a single data item. In this case, the final result on the data item depends on the order of execution of the instructions. The situation in which multiple processes access and modify same data concurrently such that the outcome depends on the order of execution, is known as *race condition*. A part of code in process that modify data (e.g., changing values of variables, updating a table, writing a file, etc.) is called the *critical section*.

To address the issue of race condition, the processes involved must be synchronized. This process synchronization ensures that only a process (or a thread) enters its critical section. A condition in which only a process at a time is allowed into its critical section among other processes that have critical sections for the same resource or shared object is called *mutual exclusion*. A simple approach to ensure mutual exclusion is to use the mutex lock. This approach requires each process to obtain a lock on the resource that they want before making any modifications to the resource. A process that does not obtain the lock must wait for another process that has the lock to finish and release that lock.

Another approach to process synchronization is to use an integer variable called *semaphore*. This approach first initializes a semaphore variable with a positive integer value, such as 1. A process that wants to enter its critical section will decrement the semaphore. If the value of the semaphore is non negative, the process could enter its critical section. Otherwise, the process should wait until

another process increments the semaphore variable when this other process completes its execution. In this lab work session, we simulate a race condition and use mutex lock and semaphore to address the race condition.

9.1 Simulating a Race Condition

To simulate a race condition, we will create two threads that modify a shared variable. The first thread attempts to increment the shared variable, while the second thread attempts to decrement the variable. First, let us create a file using the command "**nano race.c**". Then, we define the functions for the threads, the shared variable, and the main function.

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

void *increment();
void *decrement();

int shared = 1;

int main(){

   pthread_t thread1, thread2;
   pthread_create(&thread1, NULL, increment, NULL);
   pthread_create(&thread2, NULL, decrement, NULL);
   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);

   printf("Final value of the shared variable is %d\n", shared);
}
```

In the above code list, we define two functions, namely, 'increment' and 'decrement', for the two threads. Then we define and initialize the shared variable ("shared") as 1. In the main function, we define two threads, namely 'thread1' and 'thread2', and specify those threads with the 'increment' and 'decrement' functions, respectively. Next, let us specify the increment function with the following code list.

```
void *increment() {

   int x;
   x = shared;
   printf("Thread 1 reads the value of the shared variable as: %d\n", x);
   x++;
   printf("Local update by Thread 1: %d\n", x);
   sleep(1);
```

```
   shared = x;
   printf("Value of the shared variable updated by Thread 1 is: %d\n",
shared);
}
```

Thread1 will use the increment function to increment the shared variable. First, this function will read the shared variable into its local variable. Then, this function increments the local variable and sets the shared variable with the increment result. Next, let us specify the decrement function for thread2 using the code listing below.

```
void *decrement() {

   int y;
   y = shared;
   printf("Thread 2 reads the value of the shared variable as: %d\n", y);
   y--;
   printf("Local update by Thread 2: %d\n", y);
   sleep(1);
   shared = y;
   printf("Value of the shared variable updated by Thread 2 is: %d\n",
shared);
}
```

In the above code listing, the decrement function reads the shared variable into its local variable. Then, this function decrements its local variable and returns the modification result to the shared variable.

Activity 9.1

Compile and run the source code listing above and compare your result with your friends. Screenshot your result and answer the following questions.

a. Is the final result of the shared variable similar to your friends?
b. Why does this happen?
c. Which part of the source code list above is the critical section?

9.2 Process Synchronization using Mutex Lock

In this activity, we will use the mutex lock to ensure that only one thread has access to its critical section at a time. To simulate the use of mutex lock, first, let us create a copy of the source code that we wrote before and open the file for editing using the following command.

```
ubuntu@primary:~/Codes/process_sync$ cp race.c mutex.c
ubuntu@primary:~/Codes/process_sync$ ls
mutex.c  race.c  race.out
```

```
ubuntu@primary:~/Codes/process_sync$ nano mutex.c
```

Next, let us add the mutex lock variable before the main function and initialize that variable as highlighted in the code listing below.

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

void *increment();
void *decrement();

int shared = 1;
pthread_mutex_t lock;

int main(){

   pthread_mutex_init(&lock, NULL);
   pthread_t thread1, thread2;
   pthread_create(&thread1, NULL, increment, NULL);
   pthread_create(&thread2, NULL, decrement, NULL);
   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);

   printf("Final value of the shared variable is %d\n", shared);
}
```

In the above code list, we declare a mutex variable called 'lock'. Then, in the main function, we initialize that variable using the 'pthread_mutex_init' function. Next, let us add some lines of code in the increment function.

```
void *increment() {

   int x;
   printf("Thread 1 trying to acquire the lock\n");
   pthread_mutex_lock(&lock);
   printf("Thread 1 acquired the lock\n");
   x = shared;
   printf("Thread 1 reads the value of the shared variable as: %d\n", x);
   x++;
   printf("Local update by Thread 1: %d\n", x);
   sleep(1);
   shared = x;
   printf("Value of the shared variable updated by Thread 1 is: %d\n",
shared);
   pthread_mutex_unlock(&lock);
   printf("Thread 1 released the lock\n");
}
```

In the above code listing, we add some lines of code to acquire the mutex lock before reading the shared variable. At the end of the function, we add some lines of code to release the mutex lock after the modification of the shared variable is completed. We also need to add similar code lines to the 'decrement' function below.

```
void *decrement() {

   int y;
   printf("Thread 2 trying to acquire the lock\n");
   pthread_mutex_lock(&lock);
   printf("Thread 2 acquired the lock\n");
   y = shared;
   printf("Thread 2 reads the value of the shared variable as: %d\n", y);
   y--;
   printf("Local update by Thread 2: %d\n", y);
   sleep(1);
   shared = y;
   printf("Value of the shared variable updated by Thread 2 is: %d\n",
shared);
   pthread_mutex_unlock(&lock);
   printf("Thread 2 released the lock\n");
}
```

Activity 9.2

Compile and run the code list above. Screenshot the result and answer the following questions.

a. What is the final value of the shared variable?
b. Which thread obtains the lock first?
c. If the other thread first obtains the lock, what will be the final value of the shared variable?

9.3 Process Synchronization using Semaphore

In this activity, we will use semaphore for process synchronization. To this end, we create and initialize the semaphore variable and use it on the thread functions. First, let us copy the source code that we have before using the following command.

```
ubuntu@primary:~/Codes/process_sync$ cp mutex.c semaphore.c
ubuntu@primary:~/Codes/process_sync$ ls
mutex.c  mutex.out  race.c  race.out  semaphore.c
ubuntu@primary:~/Codes/process_sync$ nano semaphore.c
```

Then, let us modify the code list by adding and initializing the semaphore variable.

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

```
#include<semaphore.h>

void *increment();
void *decrement();

int shared = 1;
sem_t semaphore;

int main(){

   sem_init(&semaphore, 0, 1);
   pthread_t thread1, thread2;
   pthread_create(&thread1, NULL, increment, NULL);
   pthread_create(&thread2, NULL, decrement, NULL);
   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);

   printf("Final value of the shared variable is %d\n", shared);
}
```

In the above code listing, we first import the semaphore library. Then, we declare the semaphore variable. In the main function, we initialize the semaphore using the “sem_init” function. This function accepts three parameters. The first parameter denotes the semaphore variable. The second parameter defines the number of processes that can access this semaphore. In our case, we set this value to 0 which means that this semaphore will only be used inside this process. The third parameter initializes the semaphore value. Next, we update the increment variable to utilize the semaphore.

```
void *increment() {

   int x;
   printf("Thread 1 trying to decrement the semaphore\n");
   sem_wait(&semaphore);
   printf("Thread 1 can enter its critical section\n");
   x = shared;
   printf("Thread 1 reads the value of the shared variable as: %d\n", x);
   x++;
   printf("Local update by Thread 1: %d\n", x);
   sleep(1);
   shared = x;
   printf("Value of the shared variable updated by Thread 1 is: %d\n",
shared);
   sem_post(&semaphore);
   printf("Thread 1 increment the semaphore\n");
}
```

In the code listing above, the increment function uses the “sem_wait” function to decrement the semaphore value and check if the value is non-negative. If the value is non-negative, the thread can continue the execution of the increment function. At the end, the increment function increases the semaphore value using the 'sem_post' function. We then modify the decrement function similar to the changes we made in the decrement function.

```
void *decrement() {

   int y;
   printf("Thread 2 trying to decrement the semaphore\n");
   sem_wait(&semaphore);
   printf("Thread 2 can enter its critical section\n");
   y = shared;
   printf("Thread 2 reads the value of the shared variable as: %d\n", y);
   y--;
   printf("Local update by Thread 2: %d\n", y);
   sleep(1);
   shared = y;
   printf("Value of the shared variable updated by Thread 2 is: %d\n",
shared);
   sem_post(&semaphore);
   printf("Thread 2 increment the semaphore\n");
}
```

Activity 9.3

Compile and run the code list above. Screenshot the result and answer the following questions.

a. What is the final value of the shared variable?
b. From the final value that you obtained, which thread first obtains access to the shared variable?