CHAPTER 7
Thread and Deadlock

Learning Objectives

1. Students can develop simple programs using threads.
2. Students understand the concept of deadlock.

Introduction

A thread is the smallest unit of execution in operating systems. All threads that belong to the same process share code, data section, and other resources. A thread is a part of a process; thus, it cannot exist without a process. During execution, a thread can run concurrently. Each of the threads can handle different tasks at the same time to use CPU resources optimally. This program is called a multithread program. The operating system divides processing time not only between different applications, but also between threads within an application. The use of threads in a program is beneficial in a multiprocessor system. Each thread can run on different processors. Using threads in a program is economical, because aside from sharing resources, the switching context penalty between threads is minimal in comparison to process switch. One of the benefits of using the thread approach is to increase the responsiveness of an application. For example, in a word processor, a thread can handle grammar checking, while another thread can handle user input.

During the running of an operating system, processes are competing for resources. Some processes may require more than one resource. When two processes attempt to access the same resources, there is a possibility that a deadlock occurred. Deadlock is a situation in which some processes cannot proceed because they are waiting for the same resources. Figure 1 shows an illustration of a deadlock. In this figure, there are two processes, P1 and P2, and two resources, R1 and R2. Here, P1 requests for R1 and R2 in that order. Meanwhile, P2 requests for R2 and R1 in that order. Let us say that P1 has obtained a lock on R1 and P2 has also obtained a lock on R2. When P1 requests R2, it cannot obtain the lock because it is still used by P2. Therefore, P1 must wait until P2 finishes and unlock R2. Similarly, when P2 requests for R1, it cannot obtain the lock because it is still used by P1. Thus, P2 cannot proceed because it needs to wait until P1 finishes and unlocks R1. Because both processes are waiting for each other and cannot proceed, this condition is called a deadlock.

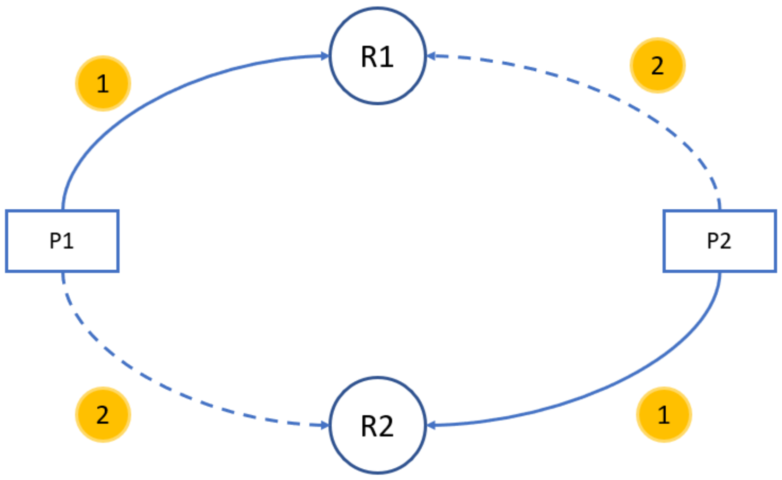

Figure 1 An illustration of deadlock

7.1 Develop a Simple Thread Program

To create a thread on Linux, we can use the “pthread_create” function from the pthread library. The parameters of the function are described below.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void arg);
```

The pthread_create function accepts four parameters. The first parameter is the buffer that will contain the identifier for the new thread. The second parameter specifies the attributes for the thread. This parameter is typically NULL unless you want to change the default settings. The third parameter specifies the function name that you want to run on the thread. The fourth parameter is the input to the function specified in the third parameter. If the thread function in the third parameter is not accepting any arguments, then the value for the fourth parameter is NULL.

Let us develop a simple thread program by creating a file using “**nano simple_thread.c**”.

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

void *thread_function(void *arg);
int i,j;


int main() {
    pthread_t a_thread;
    pthread_create(&a_thread, NULL, thread_function, NULL);
    pthread_join(a_thread, NULL);

    printf("Inside Main Program\n");
    for(j=4;j>=0;j--)
    {
         printf("%d\n",j);
         sleep(1);
    }
}

void *thread_function(void *arg) {

    printf("Inside Thread\n");
    for(i=0;i<5;i++)
    {
         printf("%d\n",i);
         sleep(1);
    }
}
```

In this code, first, we load the necessary libraries, including the pthread library. Then, we define the function and variables that will be used in the program. The function "thread_function" is the function that will be run on a thread. Afterward, in the main function, we first define the variable buffer for the thread that we will create (i.e., "a_thread"). Then, we use the "pthread_create" function to create and run a thread running the "thread_function". In the next line, we call the "pthread_join" function to wait until the "a_thread" thread terminates. The remaining lines in the main function iterate and print decreasing numbers starting from 4. Meanwhile, the "thread_function" iterates and prints increasing numbers starting from 0 to 4.

Finally, compile the code using the -lpthread flag to indicate that this program uses thread. Finally, execute the program to see how it runs.

```
~$ gcc -o simple_thread simple_thread.c -lpthread
~$ ./simple_thread
Inside Thread
0
1
2
3
4
Inside Main Program
4
3
2
```

```
1
0
```

Activity 7.1

Change the line that calls “pthread_join” in the “simple_thread.c” into a comment, compile, and run the program.

- What is the output of the program now?
- What happens?

7.2 Passing and Receiving Values to and from a Thread

We can also pass some values to a thread and receive values returned from a thread. First, let us create a file for our program using “**nano inout_thread.c**”. In this program, we will create a thread that performs addition to our two inputs and returns the result to the main program.

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

struct arg_struct
{
   int a;
   int b;
   int sum;

};

void *addition(void *arguments){

   struct arg_struct *args = arguments;
   args -> sum = args -> a + args -> b;
   pthread_exit(NULL);

}

int main(){

   pthread_t t;
   struct arg_struct *args;
   args -> a = 10;
   args -> b = 5;

   pthread_create(&t, NULL, addition, args);
   pthread_join(t, NULL);

   printf("%d + %d = %d\n", args -> a, args -> b, args -> sum);

}
```

In the source code above, after we import the necessary libraries, we create a data structure "arg_struct" to store the numbers we want to add and the result of the addition. Then, we define the function for our thread, "addition". This function adds to the two numbers, "a" and "b", and stores the result in the variable "result". Afterwards, in the main function, we declare the "t" thread buffer identifier and the "args" data structure. We then initialize the variables a and b in the data structure with the 10 and 5 values that we want to add. Then, we create and execute the thread, specifying the "addition" function as the thread function and the "args" function as the variable that we pass on to the thread function. The main function then uses pthread_join to wait until the thread finishes. Finally, the result of the operation is printed on the screen using the corresponding "args" variable. Then, compile and run this program.

```
~$ gcc -o inout_thread.out inout_thread.c -lpthread
~$ ./inout_thread.out
10 + 5 = 15
```

7.3 Deadlock

In this activity, we simulate a deadlock using multiple threads. First, we define the resources that are being competed using "pthread_mutex_t" type variables. Then, we create two threads that request these resources in a way that leads to a deadlock.

First, let us create a source code for the deadlock using "nano deadlock.c". Then, we will create the first part of the source code for the main function as follows.

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

void *function1();
void *function2();
pthread_mutex_t res_a;
pthread_mutex_t res_b;

int main() {

    pthread_mutex_init(&res_a,NULL);
    pthread_mutex_init(&res_b,NULL);

    pthread_t one, two;

    pthread_create(&one, NULL, function1, NULL);  // create thread
    pthread_create(&two, NULL, function2, NULL);
    pthread_join(one, NULL);
    pthread_join(two, NULL);
    printf("Thread joined\n");
}
```

In this first part of the source code, after we import the necessary libraries, we declare the two thread functions, “function1” and “function2”, and two resources, namely “res_a” and “res_b”. Then in the main function, we initialize the resources for the mutex lock using the “pthread_mutex_init” function. Then, we declare two identifiers for the two threads that we want to make. We then create the two threads with their corresponding identifier and function. The main function then waits until the threads finish before printing the end message.

Now, let us continue writing the first thread function that requests for both resources in a certain order as follows.

```
void *function1() {
    pthread_mutex_lock(&res_a);
    printf("Thread ONE acquired res_a\n");
    sleep(1);

    pthread_mutex_lock(&res_b);
    printf("Thread ONE acquired res_b\n");

    pthread_mutex_unlock(&res_b);
    printf("Thread ONE released res_b\n");

    pthread_mutex_unlock(&res_a);
    printf("Thread ONE released res_a\n");
}
```

The first thread function first acquires the first resource res_a and continues to acquire the second resource res_b. Then, this function releases res_b followed by releasing res_a. The second thread function is similar to this first thread function.

```
void *function2() {

    pthread_mutex_lock(&res_b);
    printf("Thread TWO acquired res_b\n");
    sleep(1);

    pthread_mutex_lock(&res_a);
    printf("Thread TWO acquired res_a\n");
    pthread_mutex_unlock(&res_a);
    printf("Thread TWO released res_a\n");

    pthread_mutex_unlock(&res_b);
    printf("Thread TWO released res_b\n");
}
```

The second thread function first acquires the second resource res_b and continues to acquire the first resource res_a. Then, this function releases res_a followed by releasing res_b. Then, let us compile and run this program.

```
~$ gcc -o deadlock.out deadlock.c -lpthread
~$ ./deadlock.out
Thread TWO acquired res_b
Thread ONE acquired res_a
```

When we run this, the program will stop until it displays the two messages above. To stop the process, press Ctrl + C.

Activity 7.2

Modify the second thread function to have the same order of accessing the resources as the first thread function.

- What is the output of the program?
- What happens?