Chapter 7

POSIX Threads

POSIX thread library is a standard thread library for C/C++. Using the POSIX thread library, we can create a new concurrent process execution flow such that our program can then handle multiple execution paths. As a result, our program would appear to do many things at the same time.

7.1 Thread Basics

A thread can be defined as a separate stream of instructions within a process. From developer point of view, a thread is simply a function or procedure, that has its own existence and runs independently from the program's main() procedure/function.

To visualize, imagine a program C program with a number of functions. This program can be run by entering the executable a.out on the command interface. Then imagine each function being scheduled by the operating system. This would be a multi-threaded program.

Unlike child processes, a thread doesn't know which thread is responsible for it's creation, neither does it maintain a list of current active threads inside a process. Within the same process, a thread may share the process instructions (text section), global data (data section) and open files (file descriptors). Each thread has a unique thread ID, set of registers and stack. These will be individual to a thread itself.

7.2 Thread Creation

A thread is created using the pthread_create() call. Just like process creation using fork(), a pthread_create() call will return certain integer numbers upon successfull completion of the call. Study the format of the code below especially the usage of the bold text.

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <XXXXX>          // Check Man
04 void *print_message(char *ptr)
05 {
06   char *message;
07   message = *ptr;
08   printf("%s \n", message);
09 }
10 int main()
11 {
12   pthread_t thread1, thread2;
13   char *message1 = "Thread 1";
14   char *message2 = "Thread 2";
15   int return_value1, return_value2;
16   return_value1 = pthread_create(&thread1,
                     NULL, print_message, *message1);
17   return_value2 = pthread_create(&thread2,
                     NULL, print_message, *message2);
18   pthread_join( thread1, NULL );
19   pthread_join( thread2, NULL );
20   exit(0);
21 }
```

Compile the above code. Compilation of multi-threaded programs are differently from the normal method. For threads, the -lpthread argument is provided as an addition.

When this program runs, there will be a total of 3 threads running "in" this process; main, thread1, and thread2. Three additional kernel level threads will be required for servicing these three user-level threads. (Remember, Linux uses 1:1 thread model).

When the thread is created using pthread_create(), the main thread proceeds executing with the remainder of instructions. Meanwhile, the newly created thread will complete executing as well. If in case the main() thread finishes executing before any of the other threads, the process will exit. Any thread which had not finished will be interrupted before its termination. To avoid this, the pthread_join() call can be used.

With this, the main() thread wil wait for successfull completion of any thread specified in the join call.

Actually, pthread_join() is the opposite of pthread_create(). Pthread_create() will split our single threaded process into two-threaded process. Pthread_join() will join back the two-threaded process into a single threaded process.

Read the manual pages and find out the answers to the following:

Q1 What is the pthread_create() and the pthread_join() calls doing?

Q2 In the pthread_create() call, what are the 4 parameters?

Q3 In pthread_create() call, the 4th parameter is used for passing a pointer to argument of a function. What will we need to do if we want to pass multiple arguments to that function?

Q4 In the pthread_join() call, what are the 2 paramters?

Q5 What is the purpose of the return_value1 and return_value2 variables? Find out the contents of both these variables using a printf function. What do they contain?

7.2.1 Passing Multiple Arguments to Thread

Multiple arguments can be passed to a thread through declaring a structure. For example,

```
struct thread_data
{
  int x;
  int y;
  int z;
}
```

Would be our thread_data structure containing members x, y, and z.

```
struct thread_data Omar;
```

Will create an instance of thread_data structure. It's name is given as Omar.

```
void *print(void *threadArg)
{
  struct thread_data *my_data;
  my_data = (struct thread_data *) threadArg;
  printf("X: %d, Y: %d, Z: &d",
         my_data->x, my_data->y, my_data->z);
}
```

Is our thread function. Here, we have declared a pointer called my_data and assign it location of argument passed to thread as input. Since this is a pointer to a structure, we will be using the arrow operator (->) instead of the dot (.) operator to access it's members.

```
int main()
{
  thread_t tid;
  omar.x = 1;
  omar.y = 2;
  omar.z = omar.x + omar.y;
  pthread_create(&tid, NULL, print,
                        (void *) &Omar);
}
```

Here, we assign the members x, y, and z some values. Instead of sending these individual members over to the thread, we send the address of the instance Omar.

7.3 Thread Termination

There are several ways of terminating threads. Summarised as follows:

- The thread "returns" from its starting routine
- Thread makes a call to pthread_exit() call
- The entire process is terminated due to call to exec() or exit()

If the main() thread finishes executing before any other thread in the process, all threads will terminate (Bullet 4). However, if main() exits using a pthread_exit() call, all other threads in the process will continue to execute. Thus the pthread_exit() call will terminate the main thread but keep on clinging to resources such as process memory space and open file descriptors.

The following code will show both thread creation and thread termination.

```
01 #include <pthread.h>
02 #include <stdio.h>

03 void *PrintHello()
05 {
06   printf("Hello World! It's me\n");
07   pthread_exit(NULL);
08 }

09 int main()
10 {
11   pthread_t threads[3];
12   int rc;
13   int t;
14   for(t=0; t<3; t++)
17   {
18     printf("In main: creating thread\n", t);
19     rc = pthread_create(&threads[t], NULL,
                      PrintHello, (void *)t);
20     if (rc)
21     {
22       printf("ERROR; return code from
                pthread_create() is %d\n", rc);
23       exit(-1);
24     }
25   }
26   pthread_exit(NULL);
27 }
```

7.4 Data Sharing between Threads

Compile and run the following code:

```
01 #include <pthread.h>
02 #include <stdlib.h>
03 #include <unistd.h>
04 #include <stdio.h>
05
06 int myGlobal = 0;
07
08 void *threadFunction()
09 {
10  int i, j;
11  for (i = 0; i<5; i++)
12  {
13   j = myGlobal;
14   j = j+1;
15   printf(".");
16   fflush(stdout);
17   sleep(1);
18   myGlobal = j;
19  }
20 }
21
22 int main()
23 {
24  pthread_t myThread;
25  int i;
26  pthread_create(&myThread, NULL,
                   threadFunction,
                   NULL);
27  for (i = 0; i < 5; i++)
28  {
29   myGlobal = myGlobal + 1;
30   printf("o");
31   fflush(stdout);
32   sleep(1);
33  }
34  pthread_join(myThread, NULL);
35  printf("\nMy Global Is: %d\n", myGlobal);
36  exit(0);
37 }
```

Fflush() is used to force a write of all user-space buffered data. Since we have specified stdout, therefore it will write it to standard output. We have two threads here again as well. The main thread has a for loop which is printing the character "o". The threadFunction thread also has a for loop which is printing the character ".". You will notice thread-scheduling in action when you see an output of:

```
localhost codes # ./thread02
o.o.o.o.o.
My Global Is: 6
```

Is everything fine with this output? Think about the myGlobal value ... Should it be 6? Or should it be 10 (2 For loops each running for 5 iterations)?

Now comment the sleep line (both or just one) and check the output. It should be 10. How come a sleep(1) can make such a difference? Here is a little explanation:

We are using the sleep() call to impose a rudimentary form of synchronization between both threads. With sleep, CPU alternates between both threads a total number of 5 times. Each time the myGlobal variable is overwritten by the threadFunction thread. Without sleep, there is just one alternation. The myGlobal variable first counts upto 5, and then it counts upto 5 again, totalling 10. But there is no synchronization between threads in this way.

If we want to impose synchronization and at the same time preserve data integrity, we would be needing something more accurate than a simple sleep() call.

7.4.1 Synchronization through Mutex

Every process has a certain portion of code which is called the "critical section" of a pro-

cess. As an example, this is the area where a process may:

1. Change shared variables
2. Write to a File
3. Use a shared resource

It would be desirable that if one process is working in this region, then another process should not be allowed to modify the contents of any shared variable within it. If, and only if, that process leaves the critical section, then another process may be allowed access to that shared region. In other words, such a region is "mutually exclusive" to one-and-only-one process at a time.

Ideally, if a process enters this region, it should lock it. Any process trying to access it in the meanwhile will not gain any access because it is locked. Once the process leaves this region, it will un-lock it, rendering it open for anybody else.

We will take this concept of critical-section and mutual exclusion and apply it to our problem in Section 7.4. Here, we apply our lock and unlock mechanism with the help of Mutexes (taken from Mutually Exclusives). If a thread is currently locked into it's critical section, another thread trying to access it will go into sleep mode. Compile and run the code given. Here, the lock is our entry section and the unlock is our exit section.

```
01 #include <pthread.h>
02 #include <stdlib.h>
03 #include <unistd.h>
04 #include <stdio.h>
05
06 int myGlobal = 0;
07 pthread_mutex_t myMutex;
08
09 void *threadFunction()
10 {
11  int i, j;
12  for (i = 0; i<5; i++)
13  {
14   pthread_mutex_lock(&myMutex);
15   j = myGlobal;
16   j = j+1;
17   printf(".");
18   fflush(stdout);
19   sleep(1);
20   myGlobal = j;
21   pthread_mutex_unlock(&myMutex);
22  }
23 }
24
25 int main()
26 {
27  pthread_t myThread;
28  int i;
29  pthread_create(&myThread, NULL,
                   threadFunction,
                   NULL);
30  for (i = 0; i < 5; i++)
31  {
32   pthread_mutex_lock(&myMutex);
33   myGlobal = myGlobal + 1;
34   pthread_mutex_unlock(&myMutex);
35   printf("o");
36   fflush(stdout);
37   sleep(1);
38  }
39  pthread_join(myThread, NULL);
40  printf("\nMy Global Is: %d\n", myGlobal);
41  exit(0);
41 }
```

7.5 Exercise

A fibonacci series is a set of numbers where every n^{th} number is the sum of the $n^{th} - 1$ and $n^{th} - 2$ numbers. The only exception to this rule is the 1^{st} and 2^{nd} numbers which are 0 and 1 respectively. The following code can find the n^{th} number in the fibonacci sequence:

```
int fib(int n)
{
  if (n<=0)
    return 0;
  else if (n==1)
    return 1;
  else
    return fib(n-1)+fib(n-2);
}
int main()
{
  int find = 40;
  printf("Element No. %d in series is: %d",
                          find, fib(find) );
  exit(0);
}
```

Note that the call to fib() function is recursive. Modify the above code so that each fib() call is implemented in a separate thread.