CSE 325- Operating Systems

Laboratory Exercise – 3

Objectives:

This lab examines aspects of threads and multithreading. Objectives of this lab are given below:

- Learn POSIX C thread API and thread library.
- Learn thread management functions – thread creation, execution and termination. Thread arguments passing and joining libraries.

What is thread? : A thread is a semi-process, which has its own stack, and executes a given piece of code. Unlike a real process, threads share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, and etc. Threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel). Pthreads library is a **POSIX C API** thread library that has standardized functions for using threads across different platforms. When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes. All threads within a process share the same address space. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication. Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:

- **Overlapping CPU work with I/O:** (for example), a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, other threads can perform CPU intensive work.
- **Priority/real-time scheduling:** tasks, which are more important, can be scheduled to supersede or interrupt lower priority tasks.
- **Asynchronous event handling:** tasks, which service events of indeterminate frequency and duration, can be interleaved. For example, a web server can transfer data from previous requests and manage the arrival of new requests. Multi-threaded applications will work on a uniprocessor system; yet naturally take advantage of a multiprocessor system, without recompiling. In a multiprocessor environment, the most important reason for using Pthreads is to take advantage of potential parallelism.

Task1: Pthread Creation and Termination

Create a file named **file3_1.c** by combining the following code segments:

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

void *kidfunc(void *p)
{
    pthread_t tid;
   tid = pthread_self();

   printf ("Kid ID is ---> %u\n", tid);
```

}

Add the following bolded lines into the body of your **main ()** function.

pthread_t kid ;

Normally when a program starts up and becomes a process, it starts with a default thread (main thread). So we can say that every process has at least one thread of control. A process can create extra threads inside the main thread or main () function. We start the main ( ) by declaring the variables for child/kid threads. ***pthread_t* is an opaque object/structure** that supports to declare a thread type object (kid). This is basically an integer used to identify the thread in the system.

After declaring the variables, we need to initialize the kid thread. We use ***pthread_create*** routine provided by the Pthreads library to accomplish the same. Following is the *pthread_create* routine.

pthread_create (&kid, NULL, kidfunc, NULL) ;

The above function requires four arguments, lets first discuss a bit on them:

- The first argument is a **pthread_t** type address. Once the function is called successfully, the variable whose address is passed as first argument will hold the thread ID of the newly created thread.
- The second argument may contain certain attributes which we want the new thread to contain. It could be priority, stack address, stack size, scope, schedule policy or etc.
- The third argument is a **function pointer**. This is something to keep in mind that each thread starts with a function and that functions address is passed here as the third argument so that the kernel knows which function to start the thread from.
- As the function (whose address is passed in the third argument above) may accept some arguments also so we can pass these arguments in form of a pointer to a void type.

  Now, **why a void type was chosen?**
  - void* can be cast to any pointer type in C. so P can point to a list containing one or more values needed by kidfunc function.
  - Similarly, the return value of kinfunc can point to a list of one or more values.

As soon as a thread is created, it will start the routine that has been assigned to it by ***pthread_create***. Here the routine/procedure name is ***kidfunc*** and it has been declared with body and prototype above.

From the code it is clear that the worker/kid thread has its own copy of the stack variables for the routine.

printf ("Parent ID is ---> %u\n", getpid( )) ;

Maximum lifetime of every thread executing in the program is that of the main thread. So, if we want that the main thread should wait until all the other threads are finished then there is a function **pthread_join()**. The pthread_join() function for threads is the equivalent of **wait()** for processes. A call to pthread_join blocks the calling thread until the thread with identifier equal to the first argument terminates.

pthread_join (kid, NULL) ;

The function above makes sure that its parent thread does not terminate until it is done. This function is called from within the parent thread and the first argument is the thread to wait on and the second

argument is the return value of the thread on which we want the parent thread to wait. If we are not interested in the return value of the main thread, we set this pointer to be **NULL**.

printf ("No more kid!\n") ;

Now complete the whole program (**file3_1.c**). Then compile, link and execute the program with the following commands:

use *#include<pthread.h>* in the program.
use *linker flag -lpthread* when linking.

Using command line arguments:

cc or gcc file3_1.c –o thread1 –lpthread
./thread1

Trace the output and write in your report.

Task3: Compile and run file3_2.c program.

I. Do the threads have separate copies of glob_data? Explain your answer with proper reasons.
II. Modify the code by taking two local variables (one inside thread function and another in main thread function). Do the value changes inside thread function effect to main or effect to thread's local data. If changing occurs then why? If not then why not? Explain your reasoning.

Task5: Compile and run file3_3.c program.

I. Write your understanding – "what does this program do?".
II. Write a program hellomany.c that will create a number N of threads specified in the command line, each of which prints out a hello message and its own thread ID. To see how the execution of the threads interleaves, make the main thread sleep for 1 second calling **sleep(1)** for every 4 or 5 threads it creates.

How does a thread return a value to the main thread?

- Understand code in file3_4.c and find your answer.

References:

1. An Introduction to Parallel Programming, Peter Pacheco.
2. POSIX Threads programming, https://computing.llnl.gov/tutorials/pthreads/
3. Tutorial points, http://www.tutorialspoint.com/cplusplus/cpp_multithreading.htm