Lab # 08

Threads

This lab examines aspects of threads and multiprocessing (and multithreading). The primary objective of this lab is to implement the Thread Management Functions:

- Creating Threads
- Terminating Thread Execution
- Passing Arguments To Threads
- Thread Identifiers
- Joining Threads

1. What is thread?

A thread is a semi-process that has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them). A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc. All these threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel).

2. What are pthreads?

Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications. In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads.

Pthreads are defined as a set of C language programming types and procedure calls. Vendors usually provide a Pthreads implementation in the form of a header/include file and a library, which you link with your program.

3. The pthreads API:

The subroutines which comprise the Pthreads API can be informally grouped into three major classes:

Thread management: The first class of functions work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)

Mutexes: The second class of functions deal with a coarse type of synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

Condition variables: The third class of functions deal with a finer type of synchronization - based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values.

Naming conventions: All identifiers in the threads library begin with pthread_

4. Thread Management Functions:

The function **pthread_create** is used to create a new thread, and a thread to terminate itself uses the function **pthread_exit**. A thread to wait for termination of another thread uses the function **pthread_join**.

Function:

Int pthread_create (pthread_t * threadhandle, pthread_attr_t *attribute, start_routine, void *arg);

Info:
Request the PThread library for creation of a new thread. The return value is 0 on success. The return value is negative on failure.

Function:

void pthread_exit (void *retval */* return value passed as a pointer */*);

Info:

This Function is used by a thread to terminate. The return value is passed as a pointer.

Function:
int pthread_join (pthread_t threadhandle void *returnvalue */* Return value is returned by ref. */*);

Info:

Return 0 on success, and negative on failure. The returned value is a pointer returned by reference. If you do not care about the return value, you can pass NULL for the second argument.

5. Thread Initialization:

Include the pthread.h library :

#include <pthread.h>

Declare a variable of type pthread_t :

pthread_t the_thread

When you compile, add -lpthread to the linker flags :

cc or gcc threads.c -o threads –lpthread

Example 1:

```
#include <stdio.h>
#include <pthread.h>
void *kidfunc(void *p)
{
printf ("Kid ID is ---> %d\n", getpid( ));
}
main ( )
{

pthread_t kid ;
pthread_create (&kid, NULL, kidfunc, NULL) ;
printf ("Parent ID is ---> %d\n", getpid( )) ;
pthread_join (kid, NULL) ;
printf ("No more kid!\n") ;

}
```

Example 2:

```
#include <stdio.h>
#include <pthread.h>
int glob_data = 5 ;
void *kidfunc(void *p)
{
printf ("Kid here. Global data was %d.\n", glob_data) ;
glob_data = 15 ;
printf ("Kid Again. Global data was now %d.\n", glob_data) ;
}
main ( )
{
```

```
pthread_t kid ;
pthread_create (&kid, NULL, kidfunc, NULL) ;
printf ("Parent here. Global data = %d\n", glob_data) ;
glob_data = 10 ;
pthread_join (kid, NULL) ;
printf ("End of program. Global data = %d\n", glob_data) ;
}
```

Example 3:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message(void * ptr)
{
int message;
message = (int*) ptr;
printf("%d \n", message);
}
int main()
{
 pthread_t thread1, thread2;
 int x=4;
 int y=5;
 int return_value1, return_value2;
 return_value1   =   pthread_create(&thread1,NULL,   print_message,
(void*) x);
 return_value2   =   pthread_create(&thread2,NULL,   print_message,
(void*) y);
 pthread_join( thread1, NULL );
 pthread_join( thread2, NULL );
exit(0);
 }
```

Example 4:

```
#include <pthread.h>
#include <stdio.h>
struct thread_data
{
     int x, y,z;
};

struct thread_data somedata;
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);
}
int main()
{
pthread_t tid;
somedata.x = 1;
somedata.y = 2;
somedata.z = somedata.x + somedata.y;
pthread_create(&tid, NULL, print, (void *) &somedata);
 pthread_join(tid, NULL );

}
```

Example 5:

```
#include <pthread.h>
#include <stdio.h>
void *PrintHello(void *threadid)
{
printf("\n%d: Hello World!\n", threadid);
pthread_exit(NULL);
}
int main()
{
pthread_t threads[3];
int rc;
int t;
for(t=0; t<3; t++)
{
printf("In main: creating thread\n", t);
rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
if (rc)
{
printf("ERROR; return code from pthread_create() is %d\n", rc);
exit(-1);
}
pthread_join(threads[t], NULL );
}
pthread_exit(NULL);

}
```

6. Synchronization through Mutex

Example 6:

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

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int myGlobal = 0;
pthread_mutex_t myMutex;
void *threadFunction()
{
int i, j;
for (i = 0; i<5; i++)
{
 j = myGlobal;
 j = j+1;
 sleep(1);
myGlobal = j;
printf("\n Child My Global Is: %d\n", myGlobal);
}
}
int main()
{
pthread_t myThread;
int i,k;
pthread_create(&myThread, NULL,
threadFunction,NULL);
 for (i = 0; i < 5; i++)
 {
   k = myGlobal;
 k = k+1;
 sleep(1);
myGlobal = k;
printf("\n Parent My Global Is: %d\n", myGlobal);
}

pthread_join(myThread, NULL);
printf("\nMy Global Is: %d\n", myGlobal);
exit(0);
}
```

Example 7:

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int myGlobal = 0;
pthread_mutex_t myMutex;
void *threadFunction()
{
```

```
int i, j;
for (i = 0; i<5; i++)
{
pthread_mutex_lock(&myMutex);
 j = myGlobal;
 j = j+1;
 sleep(1);
myGlobal = j;
pthread_mutex_unlock(&myMutex);
printf("\n Child My Global Is: %d\n", myGlobal);
}
}
int main()
{
pthread_t myThread;
int i,k;
pthread_create(&myThread, NULL,
threadFunction,NULL);
 for (i = 0; i < 5; i++)
 {
pthread_mutex_lock(&myMutex);
   k = myGlobal;
 k = k+1;
 sleep(1);
myGlobal = k;
pthread_mutex_unlock(&myMutex);
printf("\n Parent My Global Is: %d\n", myGlobal);
}

pthread_join(myThread, NULL);
printf("\nMy Global Is: %d\n", myGlobal);
exit(0);
}
```

Exercise:

Write a program using Threads to perform Array addition. You have to create threads equal to the number of elements in array each thread should perform single element addition of two arrays in parallel.