Programming Assignment # 2

Thread Creation and Execution

Objective:

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
- Passing Arguments To Threads
- Thread Identifiers
- Joining Threads

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).

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.
- 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. Most hardware vendors now offer Pthreads in addition to their proprietary API's

Why pthreads?

The primary motivation for using Pthreads is to realize potential program performance gains

- 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 both 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. This will be the focus of the remainder of this session.

Thread Management Functions:

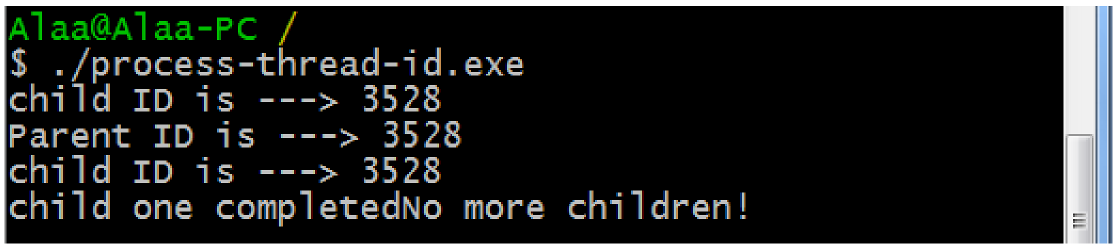
The function pthread_create is used to create a new thread, and a thread to terminate itself uses the function pthread_exit. A thread to wait for termination of another thread uses the function pthread_join.

Function	**int pthread_create (pthread_t * threadhandle,** /* *Thread handle returned by reference* */ **pthread_attr_t *attribute,** /* *Special Attribute for starting thread, may be NULL* */ **void *(*start_routine)(void *)**, /* *Main Function which thread executes* */ **void *arg** /* *An extra argument passed as a pointer* */);
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. The pthread_t is an abstract datatype that is used as a handle to reference the thread.

Function	**int pthread_join (pthread_t threadhandle,** /* *Pass 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

<u>Lab a (threads share the process ID):</u>

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

    main ( ) {
        pthread_t child1,child2 ;
        pthread_create (&child1, NULL, childfunc, NULL) ;
        pthread_create (&child2, NULL, childfunc, NULL) ;
        printf ("Parent ID is ---> %d\n", getpid( )) ;
        pthread_join (child1, NULL) ;
        pthread_join (child2, NULL) ;
        printf ("No more children!\n") ;
}
```



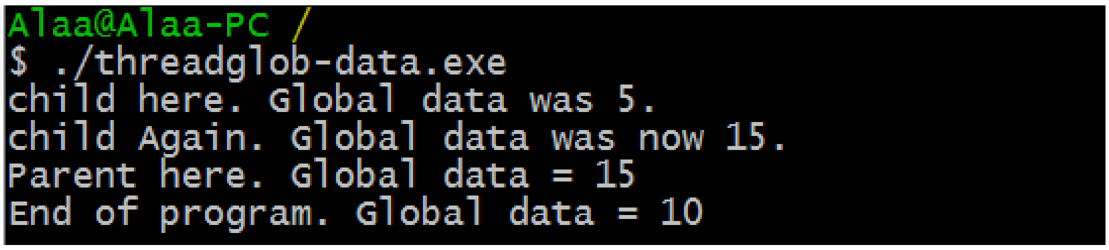
```
Alaa@Alaa-PC /
$ ./process-thread-id.exe
child ID is ---> 3528
Parent ID is ---> 3528
child ID is ---> 3528
child one completedNo more children!
```

Are the process id numbers of parent and child thread the same or different?

Lab b (Global variables effect)

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

main ( ) {
    pthread_t child1 ;
    pthread_create (&child1, NULL, childfunc, NULL) ;
    printf ("Parent here. Global data = %d\n", glob_data) ;
    glob_data = 10 ;
    pthread_join (child1, NULL) ;
    printf ("End of program. Global data = %d\n", glob_data) ;
}
```

Sample of the output:



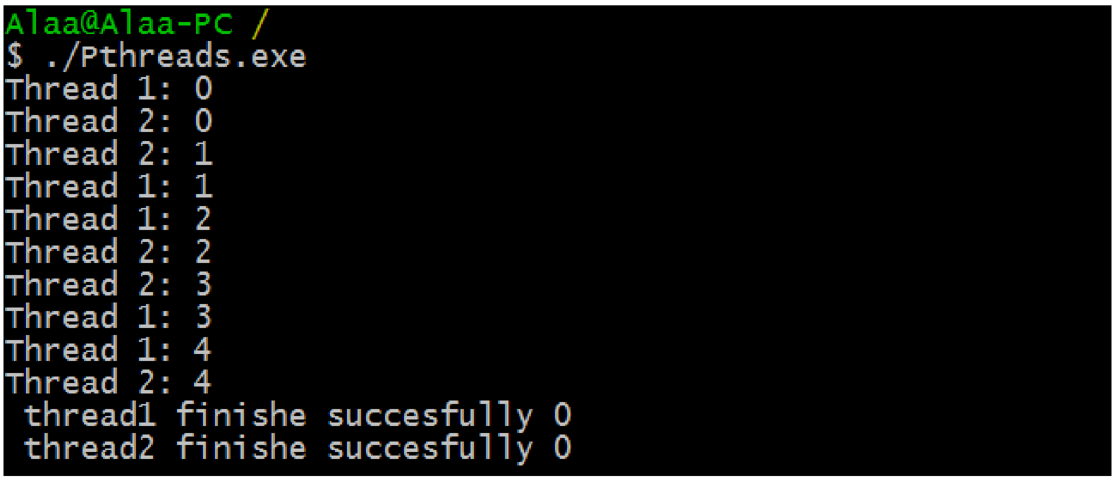
```
Alaa@Alaa-PC /
$ ./threadglob-data.exe
child here. Global data was 5.
child Again. Global data was now 15.
Parent here. Global data = 15
End of program. Global data = 10
```

Do the threads have separate copies of glob_data?

Lab c (passing parameters to threads):

In this lab, we create two threads. Each one executes a 5 steps for loop that print numbers from 0 to 4.

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

void *my_thread_process (void * arg)
{
  int i;

  for (i = 0 ; i < 5 ; i++) {
    printf ("Thread %s: %d\n", (char*)arg, i);
    sleep (1);
  }
  pthread_exit (0);
}

main (int ac, char **av)
{
  pthread_t th1, th2;


  if (pthread_create (&th1, NULL, my_thread_process, "1") < 0) {
    fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }

  if (pthread_create (&th2, NULL, my_thread_process, "2") < 0) {
    fprintf (stderr, "pthread_create error for thread 2\n");
    exit (1);
  }

  printf (" thread1 finishe succesfully %d \n" , pthread_join (th1, NULL));
    printf (" thread2 finishe succesfully %d \n" , pthread_join (th2, NULL));

}
```



```
Alaa@Alaa-PC /
$ ./Pthreads.exe
Thread 1: 0
Thread 2: 0
Thread 2: 1
Thread 1: 1
Thread 1: 2
Thread 2: 2
Thread 2: 3
Thread 1: 3
Thread 1: 4
Thread 2: 4
 thread1 finishe succesfully 0
 thread2 finishe succesfully 0
```