# Operating Systems Lab

## Threads

*By: Muhammad Ahsan*

**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 sequence of control within a process"*

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).

### a) Properties of threads:

Threads use and exist within process resources, yet are able to be scheduled by the operating system and run as independent entities largely because they duplicate only the bare essential resources that enable them to exist as executable code.
This independent flow of control is accomplished because a thread maintains its own:

1. Stack Pointer
2. Registers
3. Scheduling properties (such as policy or priority)
4. Set of pending and blocked signals
5. Thread specific data

### b) Summarize Threads

1. Exists within a process and uses the process resources.
2. Has its own independent flow of control as long as its parent process exists and the OS supports it.
3. Duplicates only the essential resources it needs to be independently schedulable.
4. May share the process resources with other threads that act equally independently (and dependently).
5. Dies if the parent process dies.

6. Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.
7. Threads do Not Share:
   a) Thread ID
   b) Set of registers, including program counter and stack pointer
   c) Stack (for local variables and return addresses)
   d) Signal mask
   e) Priority

**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 implemented with a **pthread.h** header file.
To link this library to your program use

*gcc –pthread myprogram.c –o a.out*

**3. Thread Implementation:**

Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.
**pthread_create** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.

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

**PTHREAD _ CREATE Arguments :**

1. **thread:** unique identifier for the new thread returned by the subroutine.
2. **attr:** An attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
3. **start_routine:** The C routine that the thread will execute once it is created.
4. **arg:** A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

**Thread creation steps:**

Include the pthread.h library :
#include <pthread.h>

Declare a variable of type pthread_t :

pthread_t my_thread

When you compile, add -pthread to the linker flags:
gcc –pthread threads.c -o threads

**Example 1:**

```
#include <stdio.h>
#include <pthread.h>
void *kidfunc(void *p){
        printf ("Kid ID is ---> %d\n", getpid( ));
}
int 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") ;
        return 0;
}
```

**Joining Threads:**

1. "Joining" is one way to accomplish synchronization between threads.
2. The pthread_join() subroutine blocks the calling thread until the specified threadid thread terminates.
3. The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit().
4. A joining thread can match one pthread_join() call. It is a logical error to attempt multiple joins on the same thread.

**Synchronization through Mutex:**

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.

**Example 1:**

```
#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 2:**

```
#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);
}
```