TUTORIAL: THREAD

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
* Terminating Thread Execution
* Passing Arguments To Threads
* Thread Identifiers
* Joining Threads
* Detaching / Undetaching 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.

* 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.

* 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.

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.

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. Functions to set/query condition variable attributes are also included.

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

pthread_	Threads themselves and miscellaneous subroutines
pthread_t	Thread objects
pthread_attr	Thread attributes objects
pthread_mutex	Mutexes
pthread_mutexattr	Mutex attributes objects.
pthread_cond	Condition variables
pthread_condattr	Condition attributes objects
pthread_key	Thread-specific data keys

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:	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. This pointer value can be anything so long as it does not exceed the size of (void *). Be careful, this is system dependent. You may wish to return an address of a structure, if the returned data is very large.

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

	NULL for the second argument.

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
```

Initially, threads are created from within a process. Once created, threads are peers, and may create other threads. Note that an "initial thread" exists by default and is the thread, which runs main.

Terminating Thread Execution:

```
int  pthread_cancel   (pthread_t    thread )
```

`pthread_cancel` sends a cancellation request to the thread denoted by the thread argument. If there is no such thread, **`pthread_cancel`** fails. Otherwise it returns 0.

A cancel is a mechanism by which a calling thread informs either itself or the called thread to terminate as quickly as possible. Issuing a cancel does not guarantee that the canceled thread receives or handles the cancel. The canceled thread can delay processing the cancel after receiving it. For instance, if a cancel arrives during an important operation, the canceled thread can continue if what it is doing cannot be interrupted at the point where the cancel is requested.

The programmer may specify a termination status, which is stored as a void pointer for any thread that may join the calling thread.

There are several ways in which a Pthread may be terminated:

The thread returns from its starting routine (the main routine for the initial thread). By default, the Pthreads library will reclaim any system resources used by the thread. This is similar to a process terminating when it reaches the end of main.

The thread makes a call to the **`pthread_exit`** subroutine (covered below).

The thread is canceled by another thread via the **`pthread_cancel`** routine (not covered here).

The thread receives a signal that terminates it the entire process is terminated due to a call to either the **`exec`** or **`exit`** subroutines.

Thread Attributes:

Threads have a number of attributes that may be set at creation time. This is done by filling a thread attribute object attr of type pthread_attr_t, then passing it as second argument to pthread_create. Passing NULL is equivalent to passing a thread attribute object with all attributes set to their default values. Attribute objects are consulted only when creating a new thread. The same attribute object can be used for creating several threads. Modifying an attribute object after a call to pthread_create does not change the attributes of the thread previously created.

int pthread_attr_init (pthread_attr_t *attr)

pthread_attr_init initializes the thread attribute object attr and fills it with default values for the attributes. Each attribute attrname can be individually set using the function pthread_attr_setattrname and retrieved using the function pthread_attr_getattrname.

int pthread_attr_destroy (pthread_attr_t *attr)

pthread_attr_destroy destroys the attribute object pointed to by attr releasing any resources associated with it. attr is left in an undefined state, and you must not use it again in a call to any pthreads function until it has been reinitialized.

int pthread_attr_setattr (pthread_attr_t *obj, int value)

Set attribute attr to value in the attribute object pointed to by obj. See below for a list of possible attributes and the values they can take. On success, these functions return 0.
int pthread_attr_getattr (const pthread_attr_t *obj, int *value)
Store the current setting of **attr** in **obj** into the variable pointed to by value. These functions always return 0.

<u>The following thread attributes are supported:</u>

`detachstate'

Choose whether the thread is created in the joinable state (PTHREAD_CREATE_JOINABLE) or in the detached state (PTHREAD_CREATE_DETACHED).

The default is PTHREAD_CREATE_JOINABLE. In the joinable state, another thread can synchronize on the thread termination and recover its termination code using pthread_join, but some of the thread resources are kept allocated after the thread terminates, and reclaimed only when another thread performs pthread_join on that thread.

In the detached state, the thread resources are immediately freed when it terminates, but pthread_join cannot be used to synchronize on the thread termination. A thread created in the joinable state can later be put in the detached thread using pthread_detach.

`schedpolicy'

Select the scheduling policy for the thread: one of SCHED_OTHER (regular, non-realtime scheduling), SCHED_RR (realtime, round-robin) or SCHED_FIFO (realtime, first-in first-out). The default is SCHED_OTHER. The realtime scheduling policies SCHED_RR and SCHED_FIFO are available only to processes with superuser privileges. pthread_attr_setschedparam will fail and return ENOTSUP if you try to set a realtime policy when you are unprivileged. The scheduling policy of a thread can be changed after creation with pthread_setschedparam.

`schedparam'

Change the scheduling parameter (the scheduling priority) for the thread. The default is 0. This attribute is not significant if the scheduling policy is SCHED_OTHER; it only matters for the realtime policies SCHED_RR and SCHED_FIFO. The scheduling priority of a thread can be changed after creation with pthread_setschedparam.

Thread Identifiers:

pthread_self ( )

Returns the unique thread ID of the calling thread.

The returned data object is opaque cannot be easily inspected.

pthread_equal (thread1, thread2)

Compares two thread IDs:

If the two IDs are different 0 is returned, otherwise a non-zero value is returned.

Because thread IDs are opaque objects, the C language equivalence operator == should not be used to compare two thread IDs.

Example: Pthread Creation and Termination:

```
//Lab1.c
#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") ;
}
```

Sample output

```
Parent ID is ---> 29085
Kid ID is ---> 29085
No more kid!
```

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

//Lab2.c

```
#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) ;

}
```

Sample output

```
Parent here. Global data = 5
Kid here. Global data was 10.
Kid Again. Global data was now 15.
End of program. Global data = 15
```

Do the threads have separate copies of glob_data?

---

lab6.c

/ Multithreaded C Program Using the Pthread API */*

```
#include<pthread.h>
#include<stdio.h>
int sum; /*This data is shared by the thread(s) */
void *runner(void *param);   /* the thread */

main(int argc, char *argv[]) {
         pthread_t tid;  /* the thread identifier */
         pthread_attr_t attr;     /* set of thread attributes */
         if(argc != 2)
```

```
          {
                    fprintf(stderr,"usage: a.out <integer value>\n");
                    exit();
          }

          if(atoi(argv[1]) < 0)
          {
                    fprintf(stderr, "%d must be >= 0 \n", atoi(argv[1]));
                    exit();
          }

          /* get the default attributes */
          pthread_attr_init(&attr);

          /*create the thread */
          pthread_create(&tid,&attr,runner,argv[1]);

          /* Now wait for the thread to exit */
          pthread_join(tid,NULL);
          printf("sum = %d\n",sum);
}

/*The thread will begin control in this function */
void *runner(void *param)
{
          int upper = atoi(param);
          int i;
          sum=0;
          if(upper > 0)
          {
                    for(i=1; i <= upper;i++)
                    sum += i;
          }
          pthread_exit(0);
}
```

Sample Output:

```
vlsi>gcc lab3.c -o lab3  -lpthread
vlsi>lab3   10
sum = 55
vlsi>
```

Explanation:

Above Program creates a separate thread that determines the summation of a non-negative integer. In a thread program, separate thread begin execution in a specified function . In above program it is the runner function. When this program begins, a single thread of control begins in main. After some initialization, main creates a second thread that begins control in the summer function.

All Pthread programs must include the pthread.h header file. The statement pthread_t tid declares the identifier for the thread we will create. Each thread has a set of attributes including stack size and scheduling information. The pthread_attr_t attr declaration represents the attributes for the thread. We will set the attributes in the function call pthread_attr_init(&attr). Because we did not explicitly set any attributes, we will use the default attribute provided.

A separate thread is created with the pthread_create function call . In addition to passing the thread identifier and the attributes for the thread. We also pass the name of the function where the new thread will execution, in this case runner function. Lastly , we pass the integer parameter that was provided on the command line, argv[1].

At this point , the program has two threads : the initial thread in main and the thread performing the summation in the runner function. After creating the second thread, the main thread will wait for the runner thread to complete by calling the pthread_join function. The runner thread will complete when it calls the function pthread_exit.

Multiple Threads:

The simple example code below creates 5 threads with the pthread_create( ) routine. Each thread prints a "Hello World!" message, and then terminates with a call to pthread_exit( ).

Lab4.c

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
   printf("\n%d: Hello World!\n", threadid);
   pthread_exit(NULL);
}

int main( )
{
   pthread_t   threads [NUM_THREADS];
   int rc, t;
   for(t=0; t < NUM_THREADS; t++)    {
         printf ("Creating thread %d\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_exit(NULL);
}
```

Sample output

```
vlsi> lab4
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
0: Hello World!
1: Hello World!
2: Hello World!
3: Hello World!
4: Hello World!
vlsi>
```

Difference between processes and threads:

```c
//Lab5.c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int this_is_global;

void thread_func( void *ptr );

int main( ) {
   int local_main;
   int pid, status;
   pthread_t    thread1, thread2;

   printf("First, we create two threads to see better what context they share...\n");
   this_is_global=1000;
   printf("Set this_is_global=%d\n",this_is_global);
   pthread_create( &thread1, NULL, (void*)&thread_func, (void*) NULL);
   pthread_create(&thread2, NULL, (void*)&thread_func, (void*) NULL);

   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);
   printf("After threads, this_is_global=%d\n",this_is_global);
   printf("\n");
   printf("Now that the threads are done, let's call fork..\n");
   local_main=17; this_is_global=17;
   printf("Before fork(), local_main=%d, this_is_global=%d\n",local_main, this_is_global);
   pid=fork();
   if (pid == 0) { /* this is the child */
      printf("In child, pid %d: &global: %X, &local: %X\n", getpid(), &this_is_global,
&local_main);
      local_main=13; this_is_global=23;
      printf("Child set local main=%d, this_is_global=%d\n",local_main, this_is_global);
      exit(0);
   }
   else { /* this is parent */
      printf("In parent, pid %d: &global: %X, &local: %X\n", getpid(), &this_is_global,
&local_main);
```

```
        wait(&status);
        printf("In parent, local_main=%d, this_is_global=%d\n",local_main, this_is_global);
    }
    exit(0);
}

void thread_func(void *dummy) {

    int local_thread;

    printf("Thread %d, pid %d, addresses: &global: %X, &local: %X\n",
    pthread_self(),getpid(),&this_is_global, &local_thread);
    this_is_global++;
    printf("In Thread %d, incremented this_is_global=%d\n", pthread_self(), this_is_global);
    pthread_exit(0);
}
```

Sample output

```
vlsi> lab5
First, we create two threads to see better what context they
share...
Set this_is_global=1000
Thread 4, pid 2524, addresses: &global: 20EC8, &local: EF20BD6C
In Thread 4, incremented this_is_global=1001
Thread 5, pid 2524, addresses: &global: 20EC8, &local: EF109D6C
In Thread 5, incremented this_is_global=1002
After threads, this_is_global=1002

Now that the threads are done, let's call fork..
Before fork(), local_main=17, this_is_global=17
In child, pid 2525: &global: 20EC8, &local: EFFFFD34
Child set local main=13, this_is_global=23
In parent, pid 2524: &global: 20EC8, &local: EFFFFD34
In parent, local_main=17, this_is_global=17
vlsi>
```

```
// Lab6.c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

int tot_items = 0 ;

struct kidrec {
    int data ;
    pthread_t id ;
} ;

#define NKIDS 50

void *kidfunc(void *p)
```

```
{
   int *ip = (int *)p ;
   int tmp, n ;
   tmp = tot_items ;
   for (n = 50000; n--; )
      tot_items = tmp + *ip ;

}

main ( )
{
   struct kidrec kids[NKIDS] ;

   int m ;

   for (m=0; m<NKIDS; ++m)
   {
      kids[m].data = m+1 ;
      pthread_create (&kids[m].id, NULL, kidfunc, &kids[m].data) ;
   }
   for (m=0; m<NKIDS; ++m)
      pthread_join (kids[m].id, NULL) ;

   printf ("End of Program. Grand Total = %d\n", tot_items) ;

}
```

Sample output

Run it several times until you see different output. How many times is the line?

```
       tot_items = tmp + *ip ;
```

executed? What values does *ip have during these executions?

Passing Arguments to Threads:

The **pthread_create( )** routine permits the programmer to pass one argument to the thread start routine. For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the pthread_create( ) routine. All arguments must be passed by reference and cast to (void *).

Important: threads initially access their data structures in the parent thread's memory space. That data structure must not be corrupted/modified until the thread has finished accessing it.

The following example passes a simple integer to each thread.

Example: pthread_create( ) argument passing :

```
// Lab7.c
```

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 7

char *messages[NUM_THREADS];

void *PrintHello(void *threadid)
{
   int *id_ptr, taskid;

   sleep(1);
   id_ptr = (int *) threadid;
   taskid = *id_ptr;
   printf("\n %s from thread %d \n\n", messages[taskid], taskid);
   pthread_exit(NULL);
}

int main( )
{
   pthread_t   threads[NUM_THREADS];
   int *taskids[NUM_THREADS];
   int rc, t;

   messages[0] = "English: Hello World!";
   messages[1] = "French: Bonjour, le monde!";
   messages[2] = "Spanish: Hola al mundo";
   messages[3] = "Klingon: Nuq neH!";
   messages[4] = "German: Guten Tag, Welt!";
   messages[5] = "Russian: Zdravstvytye, mir!";
   messages[6] = "Japan: Sekai e konnichiwa!";
   messages[7] = "Latin: Orbis, te saluto!";

   for(t=0;t<NUM_THREADS;t++)
   {
      taskids[t] = (int *) malloc(sizeof(int));
      *taskids[t] = t;
      printf("Creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
      if (rc)
      {
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }

pthread_exit(NULL);
}
```

Sample output

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6

English: Hello World! from thread 0
French: Bonjour, le monde! from thread 1
Spanish: Hola al mundo from thread 2
Klingon: Nuq neH! from thread 3
German: Guten Tag, Welt! from thread 4
Japan: Sekai e konnichiwa! from thread 6
Russian: Zdravstvytye, mir! from thread 5
```

Exercises

Note:

Lab Problems will be given during the lab based on material covered in this lab manual.