CS 451 – Operating Systems

Lab 05 Assignment

Due: 11:59 PM on Tuesday, 25 August 2020

You are to submit your files, to the Lab 5 folder on Blackboard by the due date and time.

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 Threads

Tutorial

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`*.

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 :

```
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. on pthread_join.

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.

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:

```c
//test1.c
#include <stdio.h>
#include <pthread.h>

void *childfunc(void *p) {
    printf ("child ID is ---> %d\n", getpid( ));
}

int main () {

    pthread_t child ;

    pthread_create (&child, NULL, childfunc, NULL) ;
    printf ("Parent ID is ---> %d\n", getpid( )) ;

    pthread_join (child, NULL) ;
    printf ("No more child!\n") ;
    return 0;
}
```

Observe the process ID numbers, are the process id numbers of parent and child thread the same or different?

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( ).

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 *).

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

#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    printf("\n%ld: Hello World!\n", (intptr_t)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
*)(intptr_t) t);
        if (rc) {
            printf("ERROR; return code from pthread_create() is
%d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

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.

Task

Write a multithreaded program **`(multi_thread.c)`** that calculates various statistical values for a list of (+ve) integers (**not more than 20**). This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, the third will determine the minimum value and the fourth will determine the median value. The variables representing the average, minimum, maximum and median values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited with the corresponding worker thread ID.

For example, suppose your program is passed the integers (from the command line as arguments)

`90 81 78 95 79 72 85`

The program will report

```
Thread 1 → The average value is 82

Thread 2 → The minimum value is 72

Thread 3 → The maximum value is 95

Thread 4 → The median value is 95
```

What to turn in

- Submit the '*multi_thread.c*' file
- A makefile (do not forget to link pthread)
- A README file