ITF22519 - Introduction to Operating Systems

Lab6: Thread Programming

Fall 2020

In this lab, you will learn how to create threads, terminate threads, wait for a thread, and write programs that use threads. Before you start, remember to `commit` and `push` your previous lab to your git repository. Then, try to `pull` the new lab:

```
$ cd ITF22519/labs
$ git pull upstream master
$ cd lab6
```

`Tasks` are what you are supposed in this lab practice. All `Tasks`, `Exercises` and your report have to be submitted to GitHub afterwards.

1 Thread Creation and Termination

To create a thread in a C program, the following statements are needed:

```
#include <pthread.h>
...
pthread_t Thread;
...
int ret = pthread_create(&Thread, NULL, (void *)startRoutine, NULL);
...
```

These lines would create a variable named `Thread` which will hold the ID of a newly created thread, courtesy of the function call `pthread_create()` and will be used to perform various functions on the thread in subsequent pthread calls.

The function call `pthread_create()` takes in as arguments:

(1) a buffer to a `pthread_t` variable (`&Thread` in this case),

(2) a pointer to a thread attribute structure (the first NULL),

(3) a function pointer to a function that the `Thread` is to perform (`void *(startRoutine)` here), and finally,

(4) a pointer to the arguments to the function that the `Thread` is to perform (the final NULL in the example).

Do note that leaving the thread attribute structure as NULL will set the attribute of the newly created thread to be the default values. The `pthread_create()` returns a value of 0 if `Thread` is successfully created, otherwise, it returns the error code.

To terminate a thread in a C program, use `pthread_exit(NULL)`.

As an example, the following piece of code that creates and terminates 10 threads each of which prints out a hello message:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>

#define NUM_OF_THREADS     10

void *PrintMessage(void *ThreadId){
      long tid;
      tid = (long)ThreadId;
      printf("Hello World from Thread #%ld!\n", tid);
      pthread_exit(NULL);
}

int main (int argc, char *argv[]){
      pthread_t threads[NUM_OF_THREADS];
      int ret;
      long i;

      for(i=0; i<NUM_OF_THREADS; i++){
            printf("Creating Thread %ld in the main() function\n", i);
            ret = pthread_create(&threads[i], NULL, PrintMessage, (void *)i);
            if (ret){
                  printf("ERROR in creating thread; return ERROR code %d\n", ret);
                  exit(-1);
            }
     }
     pthread_exit(NULL);
     return 0;
}
```

Save this code into a file called ***pthread_create.c***, and to compile the code, run gcc on the source file, but with a few extra arguments as follows:

```
gcc pthread_test.c -pthread -o pthread_test
```

Be aware the -pthread at the end of the line. This is needed to ensure that the pthread library is linked during the linking phase of gcc execution.

`Task 1:` Run the program produced by compiling ***pthread_create.c*** and answer the following questions:

1. What is expected output of the program?
2. Explain the output.

2 Waiting a Thread

As explained in the man page for `pthread_create()`, a thread that is created would terminate if the main thread returns from main, even when the created thread has not completed its task yet. To ensure that the main thread waits until the created threads complete before it continues, the function call `pthread_join()` is used. An example of how to use it is as follows:

```
#include <pthread.h>
...
pthread_t Thread;
...
int ret = pthread_create(&Thread, NULL, (void *)startRoutine, NULL);

pthread_join(Thread, NULL);
...
```

The above example will ensure that `Thread` is joined to the main thread, and the main thread will not terminate before `Thread` has finished execution of startRoutine.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>

#define NUM_OF_THREADS 10

void *PrintMessage(void *ThreadId){
       long tid;
       tid = (long)ThreadId;
       printf("Hello World from Thread #%ld!\n", tid);
       sleep (2);
}

int main (int argc, char *argv[]){
       pthread_t threads[NUM_OF_THREADS];
       int ret;
       long i;

       for(i=0; i<NUM_OF_THREADS; i++){
```

```
            printf("Creating Thread %ld in the main() function\n", i);
            ret = pthread_create(&threads[i], NULL,PrintMessage, (void *)i);
            if (ret){
                  printf("ERROR in creating thread; return ERROR code %d\n", ret);
                  exit(-1);
            }
      pthread_join(threads[i], NULL);
      }
      return 0;
}
```

`Task 2:` Answer the following questions:

1. Explain the output.
2. How the output of this program is different from that of Exercise 1.

3 Exercises

3.1 Exercise 1 (20 pts)

A C program can have several threads each of which can do different tasks. In the following program, each thread increases specific integer number into 1000. Complete the program.

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

/* The following function is run by the second thread */
void *Increase(void *a_void_ptr){
        /* Increase a to 1000 */
        int *a_ptr = (int *)a_void_ptr;

        // YOUR CODE HERE

        printf("Increasing a finished thanks to the second thread\n");
}

int main(){

        int a = 0, b = 0;

        /* Show the initial values of a and b */
        printf("Initial values: a = %d, b = %d\n", a, b);

        /* Variable for the second thread */
```

```
        pthread_t increaseThread;

        // YOUR CODE HERE

        printf("Final values: a= %d, b= %d\n", a, b);
        return 0;
}
```

Sample output

```
Initial values: a = 0, b = 0
Increasing b finished thanks to the main thread!
Increasing a finished thanks to the second thread!
Final values: a= 1000, b = 1000
```

3.2 Exercise 2 (25 pts)

The following piece of code creates two threads that print out a message special to each thread:

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

void thread1Print(void){
       printf("I am Thread 1\n");
}
void thread2Print(void){
       printf("I am Thread 2\n");
}
int main(int argc, char **argv){
      int err = 0;
      pthread_t t1;
      pthread_t t2;

      err =  pthread_create(&t1, NULL, (void *)thread1Print, NULL);
      if(err != 0){
            perror("Oop, pthread_create encountered an error!");
            exit(1);
      }
      else{
            err = 0;
      }

      err = pthread_create(&t2, NULL, (void *)thread2Print, NULL);
```

```
        if(err != 0){
              perror("pthread_create encountered an error");
              exit(1);
        }
        else{
             err = 0;
        }
        printf("I am thread 0\n");
        return 0;
}
```

Run the program produced by compiling the above code and answer the following questions:

1. What is the expected output of running this program?
2. What is the actual output of the program?

3.3 Exercise 3 (25 pts)

The behavior of the code in the above example is the result of the main thread (the thread that prints "I am thread 0") not waiting on the other two threads to complete their routine before returning from main. Add `pthread_join()` calls your program in `Exercise 1` after the creation and checking of the second thread, recompile, and run the program:

1. What is the output of the program?
2. Does it match with the expected output of the program?

3.4 Exercise 4 (30 pts)

Write a program that does matrix multiplication for matrices of size 64x64 using pthreads. Input matrices are from MatrixA.txt and MatrixB.txt file. The output matrix has to be written in a file named MatrixC.txt. If any errors are encountered during program execution, the program should print out error information and exit.

`Note`:

1. `fprintf()` can be used to write data in a text file.

2. A program that could do matrix multiplication without using pthreads will get half of the total points.

4 What To Submit

Complete all tasks under each section and submit your source files to GitHub. Complete the exercises in this lab, each exercise with its corresponding *`YourFileName.c`* file and make your `lab6_report.txt` file. After that, put all of files into the **lab6** directory of your repository. Run `git add.` and `git status` to ensure the file has been added and commit the changes by running `git commit -m "Commit Message"`. Finally, submit your files to GitHub by running `git push`. Check the GitHub website to make sure all files have been submitted.