ITF22519: Introduction to Operating Systems

Fall Semester, 2022

Lab6: Thread Programming 1

Submission Deadline: October 4th, 2022 23:59

You need to get at least 50pts to pass this lab assignment.

In this lab, you will do some practice with portable operating system interface (POSIX) Threads (or PThread) which is popular in Unix System. In particular, you will how to create a new thread, terminate a thread, wait for a thread to finish its execution, and write programs that use available API and user level thread libraries. Recall that `man page` is your best friend in Linux. Before you start, remember to `commit` and `push` your previous lab to your git repository. Then, try to `pull` the new lab:

```
$ cd OS2022/labs
$ git pull main main
$ cd lab6
```

1 Thread Creation and Termination

To create a thread, the function `pthread_create()` is needed. The syntax and parameters details of this function are given as:

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

- **pthread_t *thread**: Pointer to a `pthread_t` variable (here is thread) which is used to store a new created thread.
- **const pthread_attr_t attr:** Pointer to a thread attribute object used to set thread attributes. NULL can be used to create a thread with default arguments
- **void *(* start_routine) (void):** Pointer to thread function containing code segment which is executed by the thread.
- **void * arg:** Thread functions argument to the void.

The `pthread_create()` function starts a new thread in the calling process. The new thread starts execution by invoking `start_routine()`; `arg` is passed as the sole argument of `start_routine()`. On success, `pthread_create()` returns 0. Otherwise, it returns an error number, and the contents of *thread are undefined [Manpage]. The pieces of code below 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.

```
#include <pthread.h>
...
pthread_t Thread;
...
int ret = pthread_create(&Thread, NULL, (void *)start_routine, NULL);
// can be changed based on start_routine()
...
```

To terminate a thread in a C program, use `pthread_exit(NULL)`.
Look at the below example:

Example 1

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

#define NUM_OF_THREADS 5

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

The functions `pthread_create()` and `pthread_exit()` are declared in the header file `pthread.h`. Therefore, you need to include the header file in the top of your code. Save this code into a file called *`pthread_create.c`*. To run the above code, you need to run `gcc` in your terminal. This is almost the same as what you have done so far but you need one more extra argument `pthread`.

```
gcc pthread_create.c -pthread -o out
```

Note: Be aware of `-pthread` flag. This is needed to ensure that the `pthread` library is linked during the gcc execution.
Now, run the executable bin file `out` for **several** times. What do you see from the output?

Here is a possible output of the program:

```
ttdinh@itstud:~/OS2022/labs/lab6$ ./out
Creating Thread 0 in the main() function
Creating Thread 1 in the main() function
Hello World from Thread #0!
Creating Thread 2 in the main() function
Hello World from Thread #1!
Another message from Thread #1!

Creating Thread 3 in the main() function
Hello World from Thread #2!
Another message from Thread #2!
Creating Thread 4 in the main() function
Another message from Thread #0!

Hello World from Thread #3!
Another message from Thread #3!

Hello World from Thread #4!
Another message from Thread #4!
```

Figure 1: One example output

You may have the following observations after several runs:

- The output is **nondeterministic**. i.e, you may get different output after different run.
- There may be **no order** of threads executions. You may expect that Thread0 is executed first, then Thread1, Thread2 and so on but the program does not behave like that.
- Threads are executed in a kind of **time-interleaving** fashion. If you look at the code, you can see that each thread is supposed to execute the block of code in `PrintMessage()`. But while a thread is executing the function, it is likely that another thread is also executing that function as well. This behavior is a kind of **concurrency**, this is why Thread Programming can be called Concurrent Programming, and we have feeling that they are doing at the same time. However, the fact is that the OS can only run one thread at a given time, but we are not going to look into detail on this point:).

2 Waiting for a Thread to Finish its Executions

As explained in the `man page` for `pthread_create()`, a thread that is created would terminate if the main thread terminates even when the created thread has not completed its task yet. To ensure that the main thread waits until the created thread completes 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); //for example

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

Now, back to the problem of the **Example 1**. You can also fix the problem of **Example 1** by adding `pthread_join()` after each Thread is created:

Example 2:

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

#define NUM_OF_THREADS 5

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

}
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);
   }
   thread_exit(NULL);
   return 0;
}
```

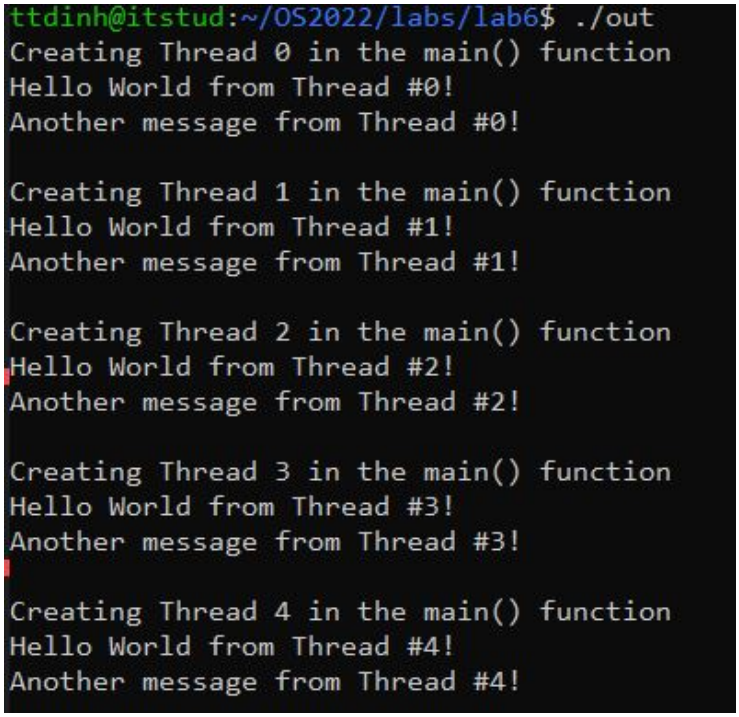

Figure 2: The output of **Example 4**

It sounds like using `pthread` is complicated. If you just want to have the output as shown in Figure 4, it may be much easier to just using `function` in C instead of `pthread`. But why do we still use `pthread`? The reason is that `thread` is a great tool to speedup the system, supposed that the developer knows how to use thread programming. There are lots of problems needed to be solved with thread programming which cannot be sorted out by simply adding `pthread_create()` or `pthread_join()`. However, this is beyond the scope of this course. In this lab, we only do simply practice with `pthread_create()` or `pthread_join()`!

3 Passing arguments to a Thread

This Section is a short guideline of how to pass argument for a Thread.
The function `pthread_create()` in Section 1 can be expressed as below:

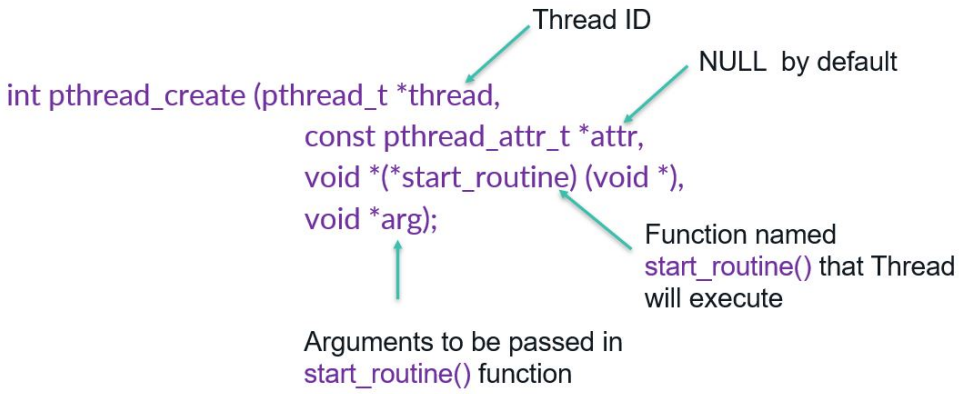

Figure 3: `pthread_create()`: syntax and parameters

Depending on how the function `start_routine()` needs arguments, you need to pass the arguments to the Thread accordingly when you create the Thread.

- If `start_routine()` does not need any passing argument, the last argument in `pthread_create` is NULL.

Example 3:

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

void *PrintMessage(void){
     printf("Message from Thread1 \n");
}

int main(int argc, char*argv[]){
    pthread_t Thread1;
    int ret;

    ret = pthread_create(&Thread1, NULL, (void*)PrintMessage,  NULL);
    if (ret!= 0){
       printf ("Thread1 creation encountered an error\n");
       exit(1);
    }
    pthread_join(Thread1, NULL);

   printf("I am the main Thread\n");

   return 0;

}
```

- If `start_routine()` needs an argument, the last argument in `pthread_create` is what will be passed in the `start_routine()` .

Example 4:

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


/* The following function is run by the second thread */
void *Increase(void *a_void_ptr){
     /* Increase a to 100 */
     int *a_ptr = (int *)a_void_ptr;

    // increasing a from 0 to 100 step by step
    for (int i = 0; i < 100; i++) {
    *a_ptr += 1;

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

int main(){

    int a = 0;

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

    /* Variable for the second thread */
    pthread_t increaseThread;

   int ret;
   ret = pthread_create(&increaseThread, NULL, (void*)Increase, &a);
   if (ret){
       printf("ERROR in creating thread; return ERROR code %d\n", ret);
       exit(-1);
   }

    pthread_join(increaseThread, NULL);
    printf("Final values: a= %d", a);
    return 0;
}
```

In **Example 1**, the function `PrintMessage()` needs a passing argument which is the ID of the Thread. Take a look at the **Example 1** again to see how the argument is passed when each thread is created in `pthread_create()`.

4 Matrix-Matrix Multiplication

This section is added because you will be asked to use Thread programming to do Matrix-Matrix Multiplication in the **Exercises**.
Matrix multiplication is one of the most basic operations that you can do with matrices. In mathematics, particularly in linear algebra, it is an operation that produces a matrix from two input matrices[1]. If you have a matrix A with size $\mathbf{M} \times \mathbf{N}$ (i.e, **M** rows and **N** columns) and another matrix B with size $\mathbf{N} \times \mathbf{P}$ (i.e, **N** rows and **P** columns), you can multiply both matrices and obtain a matrix C with size $\mathbf{M} \times \mathbf{P}$ (i.e, **M** rows and **P** columns). The product of matrices A and B is denoted as AB.

$$AB = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1N} \\ A_{21} & A_{22} & \dots & A_{2N} \\ \dots & \dots & \dots & \dots \\ A_{M1} & A_{M2} & \dots & A_{MN} \end{bmatrix} \begin{bmatrix} \begin{bmatrix} B_{11} & B_{12} & \dots & B_{1P} \\ B_{21} & B_{22} & \dots & B_{2P} \\ \dots & \dots & \dots & \dots \\ B_{N1} & B_{N2} & \dots & B_{NP} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} & \dots & C_{1P} \\ C_{21} & C_{22} & \dots & C_{2P} \\ \dots & \dots & \dots & \dots \\ C_{M1} & C_{M2} & \dots & C_{MP} \end{bmatrix} = C \tag{1}$$

where

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + \cdots + A_{iN}B_{Nj} = \sum_{k=1}^{N} A_{ik}B_{kj} \tag{2}$$

Here are some examples of matrix-matrix multiplication with small size for each matrix. For example:

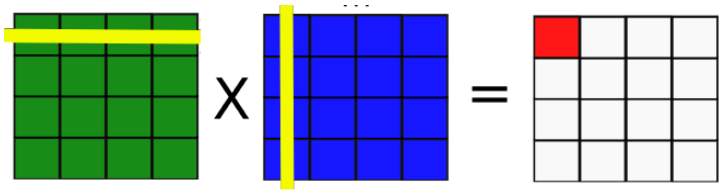

Figure 4: Matrix - Matrix Multiplication of two square matrices. The element in red of the output matrix is the product of the yellow rows from the first matrix and the yellow column of the second matrix.

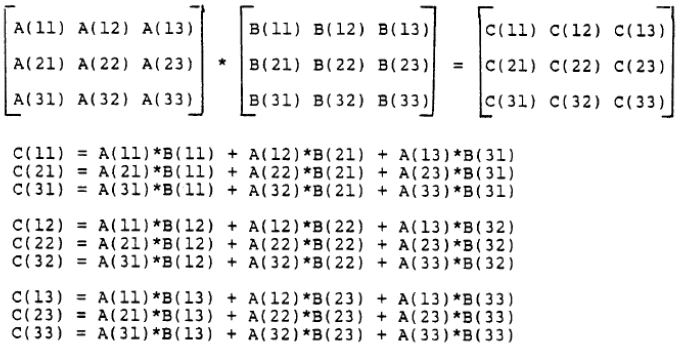

Figure 5: One case of the Equation (2).

$$\begin{bmatrix} 1 & 2 & 0 \\ 2 & 3 & 1 \\ 4 & 2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 2 & 3 \\ 4 & 2 & 1 \\ 3 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 8 & 6 & 5 \\ 15 & 12 & 10 \\ 11 & 14 & 15 \end{bmatrix} \tag{3}$$

5 Exercises

5.1 Exercise 1 (30 pts)

Using Pthread to write a C program with the following requirements:

- Create two functions named `PrintMessage1()` and `PrintMessage2()` without any input arguments. Each function prints out a different message.
- In the main function (i.e, the main thread), create two threads each of which perform one of the two functions you have created in the above step.
- Print another message in the main thread.
- Save your code in a *`Exercise1.c`* file.
- Run your code several times, copy the screenshoot in your report.
- Explain your outputs.

5.2 Exercise 2 (30 pts)

Using Pthread to write a C program with the following requirements:

- Create a function named `Deposit()` that takes an integer number as the input argument.
- Create a global variable of type `int` named `balance`. The variable `balance` has the initial value of 1000. In the main function, create another variable named `deposit` of type `int`. The value of `deposit` is user input. Next, create a thread to perform `Deposit()` function. The `Deposit()` takes `deposit` as an input argument.
- Calculate the balance after deposit money.
- Save your code in a *`Exercise2.c`* file.
- Run your code and copy the screenshoot in your report.
- **(Bonus: 10 points)** If you create another similar function, named `Widthdraw()` and then create another Thread to perform `Withdraw()`. What do you see the balance after you call two functions `Deposit()` and `Withdraw()` using Threads?

5.3 Exercise 3 (40 pts)

Make sure that you know how to do **Matrix-Matrix Multiplication**.

In the input file `A.txt`, the first element indicates the size of a square matrix. The rest of elements indicates the elements of that square matrix. The same format is applied to matrix `B.txt`. Here, the size of each matrix is 64x64. Use `pthreads` to write a C program that performs **Matrix-Matrix Multiplication** for matrix `A` and `B` where the elements of `A` and `B` are from files `A.txt` and `B.txt`. The requirements of the program are as follows:

- The main thread reads the input from `A.txt` and `B.txt` and assign their elements to the matrix `A` and `B`, respectively.
- Using 4 threads each of which is responsible for calculating 16 rows of the output matrix.
- Write the output matrix to the file `D.txt`

- Save your code in a *Exercise3.c* file.
- If your D.txt is the same as C.txt, then your output is correct.
- **(Bonus: one lab passed)** Use Pthread to do **Matrix-Matrix Multiplication** for MatrixA.txt and MatrixB.txt. You decide yourself how many Threads you will use and what each Thread will do. After you complete your code, run with the command `time ./your_executable_file` instead of `./your_executable_file` to check how long it takes to run your code. You can check if you do correctly or not by comparing your output with MatrixC.txt. :).

Note: You can reuse all Tasks and Exercises in this course.

6 What To Submit

Complete the exercises in this lab. Then, put all of files into the **lab6** directory of your repository. Make a report for each exercise. After that, 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.