ITF22519: Introduction to Operating Systems

Fall Semester, 2022

Lab7: Thread Programming 2

Submission Deadline: October 4$^{\text{th}}$, 2022 23:59

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

The topic for Thread Programming this lab is Thread Synchronization. Thread Synchronization is considered as one of the most difficult parts in Thread Programming.

In lab6, you have learned how to create, terminate a thread and to wait for a thread to finished its execution. Though the topics may be interesting, there are quite a few applications the topics can apply. For example, if we just want a thread to run a specific function, it may be much easier to just call the function doing the work instead of creating the thread to run that function. The last exercise in Lab6 is a bit more interesting where each thread is responsible for calculating a part of the output matrix. However, this is a very simple case where the threads are independent in sense that they do not have to collaborate to make the output.

In practice, we are more interested in the application that each thread solves a sub-task of the main problem and that they need to communicate with each other for the final output. In this case, multi-threads likely have to access and manipulate a shared variable or the same data. This can lead to unexpected behavior of the program. This problem can be avoided by thread synchronization i.e, we synchronize or coordinate activities of different threads in a program to get our expected output. This lab will cover how multi-threads in a C program can be synchronized by using **Mutex** and **Conditional Variables**.

Before you start this lab, 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 lab7
```

1 Mutex

When there are several threads in a program, they can access to the shared data for example the files that the process they are in is opening or shared variables. When they are reading or writing to the shared data (files or global variables), the final output can be unpredictable. This is refereed to as **race condition**.

In Lab6, we observed that the execution of several threads in a program can be **Interleaved** (thus, named **concurrence**). From a programming point of view, a thread is a block of the code. The CPU alternatively executes different instruction in different code segment. When more than one block code are trying to access the shared memory (shared files or variables), that part of the program is referred to as a **critical section** which results in the unexpected output. One way to fix the problem of critical section is to use a mechanism which allows only one thread to enter the critical section. A individual thread locks

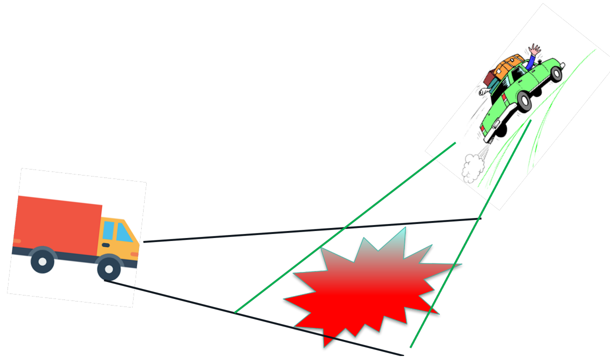

Figure 1: Two 'Threads' are trying to execute instructions in a 'critical section'.
Will an 'accident' occur :)?

the area of code, the critical section, and unlocks it once the accumulation is complete for that individual thread. To maximize performance, it is preferred that the critical section is as small as possible. The larger the critical section, the less concurrency execution among threads and thus the lower the speed of your program. To perform these locks, the following lines of code are needed:

```
pthread_mutex_t lock;
.
.
.

void *threadFunction(void *args){
.
.
.
    pthread_mutex_lock(&lock);
   //start of critical section
.
.
.
   //end of critical section
   pthread_mutex_unlock(&lock);
.
.
.
}
   int main(int argc, char** argv){
.
.
.
   err = pthread_mutex_init(&lock, NULL);
.
.
.
   err = pthread_mutex_destroy(&lock);
```

```
   return 0;
}
```

For more information about the init, lock, and unlock calls, use `man 3 pthread_mutex_init`, `man 3 pthread_mutex_lock`, and `man 3 pthread_mutex_unlock`, respectively. As can be seen above, the variable `lock` is declared as a global variable so that all threads can access to it. It is initialized in the main thread by using `pthread_mutex_init`, and the threads use it to lock critical sections using `pthread_mutex_lock`. Once the critical section is completed, it is unlocked by using `pthread_mutex_unlock`. Finally, before the program exits, destroy the mutex using `pthread_mutex_destroy` function call.

In `Exercise 3, Lab6`, each thread computes its own elements in the result matrix without need of input from another thread. Therefore, there is no reason for variables and information to be shared among multiple threads. However, if you make all threads in your code to write to the output file, the output is not what you expected. In addition, if you let each thread to read from input data files, do some calculation, and then write the results to the output files, your output is out of control :)

Example 1: In the *`Example1.c`* program, the global variable `count` is increased 10 times by thread `t1` and deceased 10 times by thread `t2` with the same amount. This means that the expected output of `count` at the end remained unchanged as its initial value which is 0. But when you run the code for several times, it is not likely that you get 0 as the output.

- What caused the discrepancy between the expected and real outputs?

The problem of the **Example 1** can be fixed by using `Mutex` explained above as follows:

```
/* This C program uses mutex to access critical section
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <pthread.h>
#include <pthread.h>

pthread_mutex_t mutex;  //global variable mutex declaration

int count;              //global variable count declaration

void Increase(void){
   int i;
   int temp;
   for( i = 0; i < 100; i++){
      pthread_mutex_lock(&mutex);
      temp = count + 10;
      usleep(1);
      count = temp;
      pthread_mutex_unlock(&mutex);
   }
}

void Decrease(void){
   int  i;
```

```
   int  temp;
   for(i = 0; i < 100; i++){
   pthread_mutex_lock(&mutex);
      temp  = count - 10;
      usleep(2);
      count = temp;
      pthread_mutex_unlock(&mutex);
   }
}

int main (int argc, char *argv[]){
    pthread_t t1;
    pthread_t t2;

    count = 0;      // global variable unitialization

    pthread_mutex_init(&mutex, NULL); // mutex unitialization

    pthread_create(&t1, NULL, (void*)&Increase, NULL);
    pthread_create(&t2, NULL, (void*)&Decrease, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("global variable count is: %d\n",count);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

- Review the code to understand how to use `Mutex`
- Run the code several times to see if the output is the same for different runs

Task 1: (In the **Exercise 2, lab6, bonus question**) In this question, the global variable `balance` is accessed and modified by 2 threads, one implements `Deposit()` function and the other implements `Withdraw()` function. According the the logic explain above, there should be the collision among two threads. What is the output of your program and what would be your explanation for the output?

2 Conditional Variables

Conditional variables are used to ensure that a thread waits until a specific condition occurs. An example of how to use a conditional variable is as follows:

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

```
int test_var;
pthread_cond_t generic_condition;
pthread_mutex_t lock;

void *genericThread0(void *args){
pthread_mutex_lock(&lock);
//do something here
pthread_cond_signal(&generic_condition);
test_var = 1;
pthread_mutex_unlock(&lock);
}

void *genericThread1(void *args){
pthread_mutex_lock(&lock);
while(test_var == 0){
pthread_cond_wait(&generic_condition, &lock);
}
//do something here
pthread_mutex_unlock(&lock);
}
.
.
.
int main(int argc, char **argv){
int test_var = 0;
.
err = pthread_mutex_init(&lock, NULL);
.
.
err = pthread_cond_init(&generic_condition, NULL);
.
.
.
err = pthread_cond_destroy(&generic_condition);
return 0;
}
...
```

For more information about `pthread_cond_init`, `pthread_cond_wait` and `pthread_cond_signal`, `pthread_cond_destroy`, use `man 3 pthread_cond_init`, `man 3 pthread_cond_wait`, and `man 3 pthread_cond_signal`, respectively.

As can be seen above, the variable `generic_condition` is declared as a global variable, similar to as `lock` is declared as a global variable in section 1. The `generic_condition` is then initialized in the main function by calling `pthread_cond_init`. `genericThread0` locks the mutex, does what is is supposed to do, and then sets the global variable `test_var` to 1 so that the function `genericThread1` can break out the loop, signals the conditional variables and then, unlocks the mutex. The `genericThread1` will attempt to lock the mutex, test the value of `test_var`, and call `pthread_cond_wait` to see if the conditional variable has been signaled. If not, the thread will block, and `pthread_cond_wait` will not return. However, according the man pages, this block does not last forever, and should be re-evaluated each time that `pthread_cond_wait` returns. Therefore, the `while` loop that srrounds the call to `pthread_cond_wait`. If the conditional variable has been signaled, then `pthread_cond_wait` would return and the thread calling it would get the mutex. The value of `test_var` would then be tested, fall through, tasks are performed and the mutex is unlock. Once all is done, remove the conditional variable using `pthread_cond_destroy`.

2.1 Exercise 3 (15 pts)

The following is a program in C

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

int count = 0;

void* Thread1PrintMessage(void* ThreadId) {
long tid;
int i;
tid = (long)ThreadId;

for (i = 0; i < 10; i++) {
printf("I am Thread 1\n");
printf("Hello World from Thread #%ld, count = %d!\n", tid, count);
sleep(2);
count++;
}
}
void* Thread2PrintMessage(void* ThreadId) {
long tid;
int i;
tid = (long)ThreadId;

for (i = 0; i < 5; i++) {
printf("I am Thread 2\n");
printf("Hello World from Thread #%ld, count = %d!\n", tid, count);
sleep(2);
count++;
}
}
int main(int argc, char** argv) {
int err = 0;
pthread_t t1;
pthread_t t2;

err = pthread_create(&t1, NULL, (void*)Thread1PrintMessage, (void*)1);
if (err != 0) {
perror("pthread_create encountered an error");
exit(1);
}
else {
```

```
err = 0;
}
err = pthread_create(&t2, NULL, (void*)Thread2PrintMessage, (void*)2);
if (err != 0) {
perror("pthread_create encountered an error");
exit(1);
}
else {
err = 0;
}
pthread_join(t1, NULL);
pthread_join(t2, NULL);
printf("I am Thread 0\n");

return 0;
}
```

- Try to analyze above code without running and estimate what the expected output would be.
- What is the actual output?
- Use what you have learnt from this lab to fix the program in order to have the expected output.

2.2 Exercise 4 (20 pts)

Write a C program `print_msg_cv.c` that prints out the message "Welcome to Østfold University College". The programs should create two threads:

- Thread 1: Prints "Welcome to"
- Thread 2: Prints "Østfold University College"

Use condition variable to synchronize the threads so that the messages are always printed in proper order.

3 Exercises

3.1 Exercise 1 (50 pts)

In the C program in *`Exercise1.c`* file, two threads try to access the critical section and modify the global variable `count`.

- Run the program several times and explain how it works.
- The global variable `count` is first initialized with 0 and then increased by `Thread1` and decreased by `Thread2` each for 10^8 times. Therefore, it is expected to be 0 when the program finishes its execution. However, the actual output is different. Use `mutex` to fix the code so that the final value of `count` is 0

3.2 Exercise 2 (50 pts)

Write a C program that uses Thread 1 to print out `Halden` and Thread 2 to print out `Fredrikstad` five times. The main Thread waits for the two threads to finish and then prints out 'Østfold University College!'. Use **condition variable(s)** to synchronize two threads so that the output is the following:

```
Fredrikstad
Halden
Fredrikstad
Halden
Fredrikstad
Halden
Fredrikstad
Halden
Fredrikstad
Halden
Østfold University College!
```

Requirements with your C program:

- Fill in the code for `print_Halden()` and `print_Fredrikstad()`, which will be executed by `Thread 1` and `Thread 2`, and `main()` as shown in Figure 2. You are not allowed to changed the order of `Thread 1` and `Thread 2`.
- Put your code on the top of code snippet for **Condition variable** in Section 1.

Code Testing:

- Your code is acceptable if it produces the correct output in all 100 runs. To test your code for 100 times, you can run it manually. If you want to test automatically in bash, you can run the following script in your Terminal. Here, `out` is the name of your executable binary output file.

```
for ((i = 0 ; i < 100 ; i++)); do
    echo $i
    ./out
done
```

```
...
//Your code for global variables (if any)
void* print_Halden () {
    //Your code for Thread 1
}
void* print_Fredrikstad () {
    //Your code for Thread 2
}
int main(int argc, char** argv) {

    //Your code for initializing any local or global variables

    pthread_t t1, t2;

    pthread_create(&t1, NULL, print_Halden, NULL);
    pthread_create(&t2, NULL, print_Fredrikstad, NULL);

   //Your code for the rest of the program
    return 0;
}
```

Figure 2: Fill out this code snippet for Exercise 2

4 What To Submit

Complete the exercises in this lab. Then, put all of files into the **lab7** 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.