ITF22519 - Introduction to Operating Systems

Lab7: Thread Programming 2

Fall 2020

You have learnt how threads are created, terminated, and waited in Lab6. Though the topic was interesting, there are quite a few applications that these ideas can apply. Many applications that use threading likely have to share variables between threads. This lab will cover the subject of how to share information and variables among threads and how race condition, conditional variables, and semaphore are used.

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 lab7
```

1 Critical Section

One way to avoid the error that occurs collision among threads is for the individual threads to lock the area of code, and unlock it once the accumulation is complete for that individual thread. This area is known as the critical section. To maximize performance, it is preferred that the critical section is as small as possible. 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`, there was no reason for variables and information to be shared between multiple threads as each thread computed its own value in the result matrix without need of input from another thread. Though great, it's not a very interesting problem to solve. For more interesting problems, there is a need to share memory between threads. The following is one example in which multiple threads have to share a common variable.

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

#define NUM_OF_THREADS     10

int count = 0;

void* PrintMessage(void* ThreadId) {
    long tid;
    int i;

    tid = (long)ThreadId;
    for (i = 0; i < 5; i++) {
        printf("Hello World from Thread #%ld, count = %d!\n", tid, count);
```

```
        count++;
        sleep(2);
    }
}

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

    pthread_create(&threads[0], NULL, PrintMessage, (void*)0);
    pthread_create(&threads[1], NULL, PrintMessage, (void*)1);
    pthread_create(&threads[2], NULL, PrintMessage, (void*)2);
    pthread_create(&threads[3], NULL, PrintMessage, (void*)3);
    pthread_create(&threads[4], NULL, PrintMessage, (void*)4);

    pthread_join(threads[0], NULL);
    pthread_join(threads[1], NULL);
    pthread_join(threads[2], NULL);
    pthread_join(threads[3], NULL);
    pthread_join(threads[4], NULL);

    return 0;
}
```

1.1 Exercise 1 (15 pts)

- What is the expected output of the above program?
- Run the code and explain the output.
- What caused the discrepancy between the expected and real outputs?

1.2 Exercise 2 (20 pts)

Fix the discrepancy of `Exercise 1` by using `pthread_mutex_lock`, `pthread_mutex_unlock`, and related functions.

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);
          count++;
          sleep(2);
     }
}
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);
             count++;
             sleep(2);
        }
}
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.
- What would be the difference between the output of this exercise compared to that of Exercise 2?

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 Semaphore

Semaphores perform a similar task to conditional variables, and they are slightly easier to use. Semaphores come it two flavors, Named, and Unnamed. The differences between the two are in how they are created, and destroyed. For simplicity, the unnamed flavor of semaphores will be covered in this handout. To use semaphores, the following function calls and includes are needed:

```
#include <semaphore.h>

sem_t semaphore;
.
.
.
void *genericThread0(void *args){
    pthread_mutex_lock(&lock);
    err = sem_wait(&semaphore);
 ...
    //do something here
    err = sem_post(&semaphore);
 ...
    pthread_mutex_unlock(&lock);
}

void *genericThread1(void *args){
    err = sem_wait(&semaphore);
    pthread_mutex_lock(&lock);
 ...
    //do something here
err = sem_post(&semaphore);
...
    pthread_mutex_unlock(&lock);
}
```

```
int main(int argc, char **argv){
 .
 .
 .
    err = sem_init(&semaphore, 0, 1);
 .
 .
 .
    err = sem_destroy(&semaphore);
 ...
    return 0;
}
```

For more information on the init, wait, post, and destroy functions, consult `man 3 sem_init`, `man 3 sem_wait`, `man 3 sem_post`, and `man 3 sem_destroy`, respectively. Note that the calls to `pthread_mutex_lock` and `pthread_mutex_unlock` do not necessarily have to be where they are shown in the above code. The reason is that the mutex lock does not have to occur before the semaphore wait, and the mutex unlock doesn't have to occur after the semaphore post.

For similar reasons to `lock` and `generic_condition`, `semaphore` is declared as a global variable. It is initialized in main function by using `sem_init` with the value for `pshared` set to 0 (meaning the semaphore is only shared between the threads of the current process). and it's initial value will be '1'(last argument to `sem_init`).

3.1 Exercise 5 (20 pts)

Write a program `print_msg_sm.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 semaphore to synchronize the threads so that the messages are always printed in proper order.

3.2 Exercise 6 (20 pts)

Write a program `print_AB_sm.c` that has two threads:

- Thread 1 = while true do print A
- Thread 2 = while true do print B

Add semaphores such that at any moment the number of A or B differs by at most 1. The solution should allow strings such as: ABBAABBABA

4 What To Submit

Complete the exercises in this lab, each exercise with its corresponding *YourFileName.c* file and make your `lab7_report.txt` file. After that, put all of files into the **lab7** 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.