ITF22519: Introduction to Operating Systems

Fall Semester, 2021

Lab7: Thread Synchronization

Submission Deadline: October 14th, 2021 23:59

In Lab6, you have learnt how a thread is created, terminated and waited by the calling thread. 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 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. They do not have to collaborate to make the output.

In practice, we are more interested in the case that each thread solves a sub-task of the main problem and that they need to communicate with each other. In this case, multi-threads likely have to access and manipulate shared variables. This can lead to unexpected behavior of the program. This problem can be avoided by thread synchronization. This lab will cover how multi threads in a C program can be synchronized by using race condition, conditional variables and semaphores.

Before you start, remember to `commit` and `push` your previous lab to your git repository. Then, try to `pull` the new lab:

```
$ cd Introduction2OS/labs
$ git pull upstream main
$ cd lab7
```

This lab is mandatory. In order for exercise section to be more readable, I made a seperate file for it. Please check `Lab7_MandatoryAssignment.pdf` file for exercises you are supposed to complete.

1 Mutex

One way to avoid collision among threads is for each thread to lock the area of code and unlock once the execution is completed for that individual thread. This area is refered to as the critical section. To maximize performance, it is preferred that the critical section is as small as possible. To perform these locks, use the following piece of codes:

```
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;
}
```

In the above code snippet, the variable `lock` is declared as a global variable so that all threads can access to it and manipulative its value. The global variable `lock` is initialized in the main thread by using `pthread_mutex_init`. Other threads use it to lock the critical section using `pthread_mutex_lock`. Once the critical section is completed, it is unlocked by using `pthread_mutex_unlock`. Finally, before the program exits, it is neccessary to destroy `mutex` using `pthread_mutex_destroy` function call. For more information about `init`, `lock` and `unlock` calls, use `man 3 pthread_mutex_init`, `man 3 pthread_mutex_lock`, and `man 3 pthread_mutex_unlock`, respectively.

Now, let's do simple practice lock and unlock. In the following program, multiple threads are trying to access and modify a global variable `count`.

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

#define NUM_OF_THREADS     5

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

```
}

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;
}
```

Task 1:

- What would be the expected output of the above program?
- Run the code and explain the output.
- If the expected output is different from the actual output, what caused the discrepancy between them?

Task 2: Fix the discrepancy of **Task 1** by using `pthread_mutex_lock`, `pthread_mutex_unlock`, and related functions.
Note: This discrepancy in fact can be fixed in a much simpler way.

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;
}
...
```

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`.

For more information about `init`, `wait`, `signal` and `destroy`, use `man 3 pthread_cond_init`, `man 3 pthread_cond_wait`, and `man 3 pthread_cond_signal`, respectively.

Example 1 The program below prints out the message "Welcome to Østfold University College" by using 2 threads:

- Thread 1: Prints “Welcome to”
- Thread 2: Prints “Østfold University College”

The program uses condition variable to synchronize the threads so that the messages are always printed in proper order. Run the program and analyze the code yourself.

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

int signal = 0;
pthread_cond_t generic_condition;
pthread_mutex_t lock;

void* Thread1PrintMessage(void* ThreadId) {
     pthread_mutex_lock(&lock);

     printf("Welcome to ");

     pthread_cond_signal(&generic_condition);
     signal = 1;
     pthread_mutex_unlock(&lock);
}

void* Thread2PrintMessage(void* ThreadId) {
     pthread_mutex_lock(&lock);
     while(signal == 0) {
          pthread_cond_wait(&generic_condition, &lock);
     }

     printf("Østfold University College\n");

     pthread_mutex_unlock(&lock);
}

int main(int argc, char** argv) {
     int err = 0;
     pthread_t t1;
     pthread_t t2;

     err = pthread_mutex_init(&lock, NULL);
     if (err != 0) {
          perror("pthread_mutex_init encountered an error");
      }
     else {
          err = 0;
     }
 err = pthread_cond_init(&generic_condition, NULL);
```

```
    if (err != 0) {
          perror("pthread_cond_init encountered an error");
    }
    else {
          err = 0;
    }
 err = pthread_create(&t1, NULL, (void*)Thread1PrintMessage, NULL);
    if (err != 0) {
          perror("pthread_create encountered an error");
          exit(1);
    }
    else {
          err = 0;
    }
 err = pthread_create(&t2, NULL, (void*)Thread2PrintMessage, NULL);
    if (err != 0) {
          perror("pthread_create encountered an error");
          exit(1);
    }
    else {
         err = 0;
    }

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

    err = pthread_mutex_destroy(&lock);
    if (err != 0) {
         perror("pthread_mutex_destroy encountered an error");
    }
    else {
         err = 0;
    }
 err = pthread_cond_destroy(&generic_condition);
    if (err != 0) {
         perror("pthread_cond_destroy encountered an error");
    }
    else {
         err = 0;
    }
    return 0;
}
```

3 Semaphores

Semaphores perform a similar task to conditional variables, and they are slightly easier to use. 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, use `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`).

Example 2 The program below prints out the message "Welcome to Østfold University College" by using 2 threads:

- Thread 1: Prints “Welcome to”
- Thread 2: Prints “Østfold University College”

The program uses semaphore to synchronize the threads so that the messages are always printed in proper order. Run the program and analyze the code yourself.

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

sem_t semaphore;

void *Thread1Print(void *ThreadID){
     printf("Welcome to ");
     sem_post(&semaphore);
}
void *Thread2Print(void *ThreadID){
     sem_wait(&semaphore);
     printf("Østfold University College\n");
}

int main(){
    pthread_t t1, t2;
    sem_init(&semaphore, 0, 0);

    pthread_create(&t1, NULL, Thread1Print, (void*)1);
    pthread_create(&t2, NULL, Thread2Print, (void*)2);

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

    sem_destroy(&semaphore);

}
```

Note: While using semaphores:

- In this lab, sleep() or usleep() functions, if any, should be used only for illustration purpose. If you have to use these functions to synchronize threads, you do not use semaphores in the correct way.
- You may experience that threads are waiting for each other forever, no one will be executued. This happens if you do not use semaphores in the correct way. This phenomenon can be refered to as deadlock state.

4 What To Submit

Follow submission instruction in `Lab7_MandatoryAssignment.pdf` file.