Programming Assignment 2 (Process Synchronization) [**25 pts + Bonus 5 pts**]

COMP 4270: Operating Systems

Fall 2018

Due: 10/31/2018

Objectives

By completing this programming assignment, you will be familiarized with POSIX threads, thread communication, and synchronization.

Overview

In this assignment, you will be given code for a multi-threaded game simulation. The problem is that there is a number of race conditions (critical sections), and the threads are not synchronized. As a result, the program does not work properly (e.g., terminating prematurely) or even crashes. You are tasked to identify critical sections of the code and implement mutual exclusion using a process synchronization tool – mutex lock.

The Game Simulation

The provided code implements a game simulation. In the game, there are players of two teams. Each player is represented as a thread. These players (threads) run around the field randomly and try to pass and steal balls. When the game is started, each team is given a ball – thus there are two balls. Each player runs around the field randomly and checks if there is any adjacent player. If he has a ball and there is an adjacent player of the same team, he will pass the ball to the adjacent player. On the other hand, if he does not have a ball and finds an adjacent player of the opposite team having the ball, he will try to steal the ball from the adjacent player. This pass or steal may fail randomly.

Task 1 [5 pts]

Download, read, and execute the code for the game from eCourseWare. When executing the code, don't forget to link with the pthread library, *e.g.,*

```
gcc -O2 hw2.c -lpthread
```

Read the code carefully and try to find out which parts of the code are critical sections.

Task 2 [5 pts]

Once you understand the code and find the critical sections, let’s implement mutual exclusion for the critical sections by adding mutex locks. Specifically, add mutex locks to avoid race conditions in the implementation. To keep it simple (at first), use one big lock for everything but the random number generator, and use a separate lock for the random-number generator. In particular, it should be possible for one player thread to generate a random number while a different player thread manipulates the field.

See below example code that illustrates how to use mutex locks. Focus on the red-colored lines and learn where to put the code for initializing, using, and destroying the mutex lock.

```
#include<stdio.h>

#include<string.h>

#include<pthread.h>

#include<stdlib.h>

#include<unistd.h>



pthread_t tid[2];

int counter;

pthread_mutex_t lock; // A mutex lock. Note that this is a global variable so the

                      // threads can share.



void* doSomeThing(void *arg)

{

    pthread_mutex_lock(&lock);
    unsigned long i = 0;

    counter += 1;

    printf("\n Job %d started\n", counter);



    for(i=0; i<(0xFFFFFFFF);i++);



    printf("\n Job %d finished\n", counter);



    pthread_mutex_unlock(&lock);



    return NULL;

}



int main(void)

{

    int i = 0;

    int err;

    // You must initialize the mutex lock before using it!

    if (pthread_mutex_init(&lock, NULL) != 0)

    {

        printf("\n mutex init failed\n");
        return 1;

    }



    while(i < 2)

    {

        err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);

        if (err != 0)

            printf("\ncan't create thread :[%s]", strerror(err));

        i++;

    }



    pthread_join(tid[0], NULL);

    pthread_join(tid[1], NULL);

    // You must destroy the mutex lock after using it.

    pthread_mutex_destroy(&lock);



    return 0;

}
```

Your resulting implementation should run without crashing or other undesireable behavior (such as losing track of a player or making more than 500000 moves). Run your program a number of times (10 to 20 times) and make sure that your program does not crash and does not give any error messages. Then, you are good.

Task 3 [5 pts]

Random-number generation doesn't need to be shared among processes. Move the random-number state into each player, so that each can generate random numbers without synchronizing.

```
/* Random-number state: */

static rand_state_t r;
```

Don’t forget to remove the lock that you used for the random number generator in Task 2 since we do not need it as we moved the random number generator into each player.

Task 4 [10 pts]

The main thread in the simulation constantly spins, waiting for the epoch counter to increase. That's bad, because it wastes CPU. It's better to send some sort of notification to the main thread when the epoch is increased, and let the main thread otherwise sleep.

One simple form of waiting and notification is writing and reading from a **pipe**. Change the program so that when the epoch counter is increased, a byte is written into a pipe. Then the main thread can block by attempting to read from the pipe to wait for the epoch counter to increase.

Here is an example that illustrates how to use the pipe:

https://linuxprograms.wordpress.com/2008/01/23/piping-in-threads/

Challenge: Task 5 [Bonus 5 pts]

When a player discovers an adjacent player and tries to pass or steal, then a little contest starts (which can take a while) to determine whether the pass/steal succeeds or fails. Some lock must be held during the contest, so that the other player doesn't move. The field position isn’t going to change, however, so the lock for a contest doesn’t have to be the big lock for the whole field.

For finer-grained locking during a contest, associate a mutex lock with each player. Before a contest, lock the two players, then release the field lock, so that other players can continue to move. You’ll probably also want a separate lock for the pass/steal statistic counters, since they may be updated after a contest.

Keep in mind that when a process acquires multiple locks, it should acquire them in a globally agreed-on order. Otherwise, two processes can deadlock while trying to acquire the same locks in different orders. The original code assigns each player a unique rank, which can be used to order the player locks.

Evaluation criteria

- Your assignment will be evaluated based on the following:

  Documentation 10% - your code should be easy to read and well commented. For each function used in your program, the use of function, its parameters, and return values should be well described.

  Compilation 20% - your program should compile with no errors and/or warnings (base points)

  Correctness 70% - To grade your work, in addition to running your code, we will also look at your code to see if you have identified critical sections correctly and implemented mutual exclusion correctly.

References

[1] CS 5460 Operating Systems at University of Utah (https://www.cs.utah.edu/~mflatt/past-courses/cs5460)