

HW#2. Multithreading and Synchronization

Overview

In this homework, you will have the chance to use threads and see how to get them synchronized. In case you are not familiar with the use of Pthread (POSIX Threads) APIs, following is an example showing how to create threads with `pthread_create`.

Sample code:

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

pthread_mutex_t mutex;

void* hello(void*) {
    pthread_mutex_lock(&mutex);
    printf("%ld says hello\n", pthread_self());
    pthread_mutex_unlock(&mutex);
}

int main () {
    pthread_t thread[3];
    int i;
    for (i = 0; i < 3; i++) {
        pthread_create(&thread[i], NULL, hello, NULL);
    }

    for (i = 0; i < 3; i++) {
        pthread_join(thread[i], NULL);
    }
    return 0;
}
```

Sample output:

```
140324646663936 says hello
140324638271232 says hello
140324629878528 says hello
```

For more details on Pthread APIs, please refer to the following webpages.

Pthread: <https://computing.llnl.gov/tutorials/pthreads/#CreatingThreads>

Mutex: <https://computing.llnl.gov/tutorials/pthreads/#MutexLocking>

Semaphore: <http://www.amparo.net/ce155/sem-ex.html>

Spinlock: https://docs.oracle.com/cd/E26502_01/html/E35303/ggecq.html

For the homework, you need to complete three tasks, which are outlined as follow.

Task 1 Parallel counter (mutex, semaphore, spinlock)

```
class Counter
{
    int value;

    public:
        Counter() { value = 0;}

        void Increment()
        { value++; }

        void Print()
        { cout<<value;}
};

Counter x;

void* ThreadRunner(void*)
{
    int k;
    for (k = 0; k < 100000000; k++)
        x.Increment();
}

pthread tid[3];
int i;
for ( i = 0; i < 3; i++)
    pthread_create(&tid[i], NULL, ThreadRunner, 0);

for ( i = 0; i < 3; i++)
    pthread_join(tid[i], NULL);

x.Print();
```

Figure 1. Parallel counter x

As shown in Figure 1, we use three threads to increment the value of a shared counter `x` in parallel. The initial value of `x` is 0. Each thread makes 100,000,000 calls to `x.Increment()`. Intuitively, the `x.Print()` at the very end should show 300,000,000, but most likely that won't be the case when running on your computer.

For the `x.Print()` to show the value of 300,000,000, you have to add synchronization code to prevent concurrent invocations of `x.Increment()`. Please use **pthread_mutex**, **semaphore**, **pthread_spinlock**, and **homemade_spinlock** respectively for the synchronization. Also, please compare the execution time of the parallel counter with respect to each of the synchronization mechanisms.

For each synchronization method, use the following api to accomplish the

work.

pthread_mutex:

```
pthread_mutex_lock(pthread_mutex_t *mutex);  
pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Semaphore:

```
sem_init(sem_t *sem, int pshared, unsigned int value);  
sem_wait(sem_t *sem);  
sem_post(sem_t *sem);
```

pthread_spinlock:

```
pthread_spin_init(pthread_spinlock_t *lock, int pshared);  
pthread_spin_lock(pthread_spinlock_t *lock);  
pthread_spin_unlock(pthread_spinlock_t *lock);
```

Homemade_spinlock:

initialize spinlock to 0 before using it

```
homemade_spin_lock(int *spinlock_addr);  
homemade_spin_unlock(int *spinlock_addr)
```

```
void homemade_spin_lock(int *spinlock_addr) {  
  
    asm(  
        "spin_lock: \n\t"  
        "xorl %%ecx, %%ecx \n\t"  
        "incl %%ecx \n\t"  
        "spin_lock_retry: \n\t"  
        "xorl %%eax, %%eax \n\t"  
        "lock; cmpxchgl %%ecx, (%0) \n\t"  
        "jnz spin_lock_retry \n\t"  
        : : "r" (spinlock_addr) : "ecx", "eax" );  
    }  
  
void homemade_spin_unlock(int *spinlock_addr) {  
  
    asm(  
        "spin_unlock: \n\t"  
        "movl $0, (%0) \n\t"  
        : : "r" (spinlock_addr) : );  
    }  
}
```

Figure 2. Homemade spinlock

Task2 Estimate pi with Monte Carlo Method

Here we are going to use the “Monte Carlo” method to estimate the value of π (pi). This can be done by inscribing a circle of radius R on a square with side length $2R$. The area of the circle will be πR^2 and the area of the square will be $(2R)^2$. Now randomly pick N points inside the square, the number of points in the circle will be approximately $N\pi/4$.

Choose R as 1. Randomly pick the coordinates of the points x and y between -1 and 1 . If $x^2 + y^2 \leq 1$, then this point is in the circle. The estimated π will be $4 * \text{number_of_points_in_circle} / \text{total_number_of_points}$.

$$\frac{\text{number_of_points_in_circle}}{\text{total_of_points}} = \frac{\text{area_of_the_circle}}{\text{area_of_the_square}} = \frac{\pi}{4}$$

Write a program that takes one command-line argument indicating how many points should be picked. When you get π print it in a line and the format is "pi estimate = **your_pi**". For example, argument “10000000” means the `total_number_of_points` is 10000000.

Implement this task in two methods

First method (lock free): implement Monte Carlo method without using any synchronizing method. One way is that each thread maintains its own counters for `number_of_points_in_circle`. Count the total number of points by summing the counters of each thread at the very end.

Second method (use lock): implement Monte Carlo method using synchronizing method. Use only one `number_of_points_in_circle` counter and one `total_number_of_points` counter.

Task3 Cross the intersection

There are cars coming from four directions (North, East, South, and West) at an intersection as shown in Figure 3. There are three possible actions for a car, which are **acquiring a lock (succeed or fail)**, **crossing the intersection** and **releasing a lock**.

1. Acquiring a lock (succeed or fail):

A car must acquire two locks in front of it before it can cross the intersection. If a car waiting at the intersection is able to acquire a lock, it must try it. A car cannot stay idle. The acquire action will fail if the lock is hold by another car. Note that the acquire action never blocks. It either returns with success (lock acquired) or failure (lock not acquired).

2. Crossing the intersection:

After a car has acquired the locks, it can cross the intersection. Your program should print out “**DIRECTION NUMBER** leaves at **X**” in one line for each car crossing the intersection. **DIRECTION** can be N, E, S, or W, which indicates the direction which the car comes from. **NUMBER** is the number of the cars that have crossed the intersection from the same **DIRECTION**. **X** is the clock time when the car crosses the intersection.

3. Releasing locks:

When a deadlock happens, one car will release its locks.

After a car has crossed the intersection, it releases the locks it acquired. Other cars may acquire the released locks in the same clock time.

Each action takes one time unit. A failed lock acquiring action also takes one time unit. The clock time starts from 1.

A car cannot release its locks unless a deadlock happens or until it has crossed the intersection. Write a deadlock happens, your program should print out “A DEADLOCK HAPPENS at **X**”, where **X** is the time unit when the deadlock happens. After that, resolve the deadlock by letting one of the car holding locks release all its locks.

Program requirements:

Create a thread to simulate cars from each direction, so you will create four threads at least to simulate cars from four directions. The program takes four numbers. The four numbers correspond to the numbers of cars waiting in line at the four directions (follow the order of N, E, S, W). For example, argument “1 0 2 3” means there are one car from N, zero car from E, two car from S, and three car from W.

For example, the argument “1 0 0 0” means that there is a car from N called N 0. N 0 has to get two lock which are lock_0 (step 1) and lock_2 (step 2). Once N 0

gets the two locks, it can cross the intersection (step 3) and prints out “N 0 left at 3”.

	N		
	lock_0	lock_1	E
W	lock_2	lock_3	
		S	

Figure 3. Cars crossing an intersection

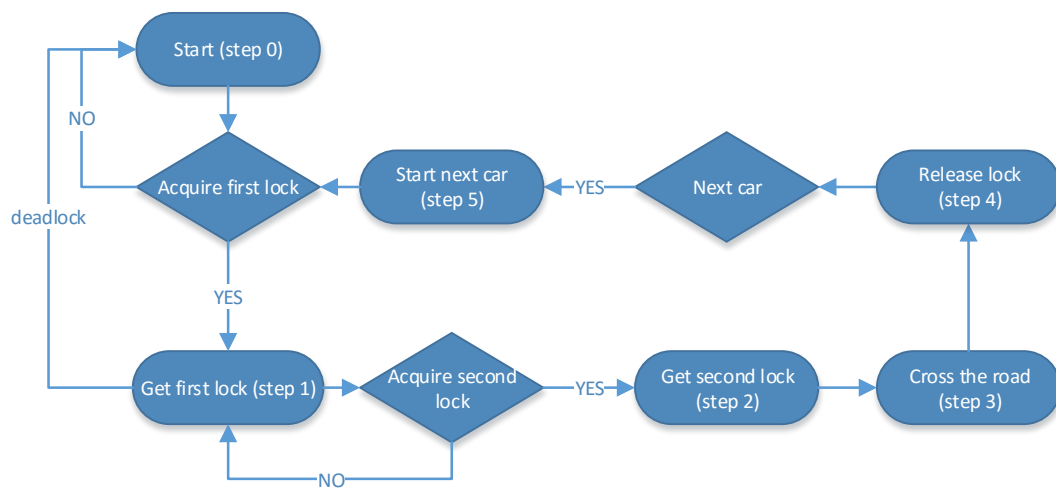


Figure 4. State transition diagram for cars crossing the intersection

Example for argument “2 0 0 0”:

- 1: N 0 acquire lock_0.
- 2: N 0 acquire lock_2.
- 3: N 0 crosses the road and print out “A 0 left at 3”.
- 4: N 0 releases lock_0 and lock_2.
- 5: N 1 acquire lock_0.
- 6: N 1 acquire lock_2.
- 7: N 1 crosses the road and print out “A 1 left at 7”.
- 8: N 1 releases lock_0 and lock_2.

```

N 0 leaves at 3
N 1 leaves at 7
  
```

Figure 5. Example for argument “2 0 0 0”

Example for argument "1 1 0 0": (possibility 1)

1: N 0 acquire lock_0.

E 0 acquire lock_1.

2: N 0 acquire lock_2.

E 0 acquire lock_0 (fail)

3: N 0 crosses the road and print out "A 0 left at 3"

E 0 acquire lock_0 (fail)

4: E 0 acquire lock_0 (fail)

N 0 releases lock_0 and lock_2.

5: E 0 acquire lock_0.

6: E 0 crosses the road and print out "E 0 left at 6"

7: E 0 releases lock_0 and lock_1.

```
N 0 leaves at 3
E 0 leaves at 6
```

Figure 6. Example for argument "1 1 0 0"

Example for argument "1 1 0 0": (possibility 2)

1: N 0 acquire lock_0.

E 0 acquire lock_1.

2: N 0 acquire lock_2.

E 0 acquire lock_0 (fail)

3: N 0 crosses the road and print out "A 0 left at 3"

E 0 acquire lock_0 (fail)

4: N 0 releases lock_0 and lock_2.

E 0 acquire lock_0.

5: E 0 crosses the road and print out "E 0 left at 5"

6: E 0 releases lock_0 and lock_1.

```
N 0 leaves at 3
E 0 leaves at 5
```

Figure 7. Example for argument "1 1 0 0"

Example for argument "1 1 1 1": (deadlock)

1: N 0 acquire lock_0

E 0 acquire lock_1

W 0 acquire lock_2

S 0 acquire lock_3

2: N 0 acquire lock_2 (fail)

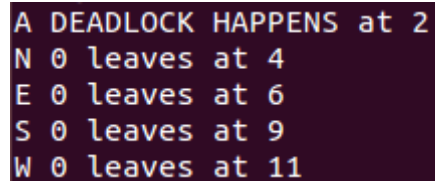
E 0 acquire lock_0 (fail)

W 0 acquire lock_3 (fail)

S 0 acquire lock_1 (fail)

Print out "A DEADLOCK HAPPENS at 2"

The rest depends on your deadlock handler. The graph below is just one possible output.



```
A DEADLOCK HAPPENS at 2
N 0 leaves at 4
E 0 leaves at 6
S 0 leaves at 9
W 0 leaves at 11
```

Figure 8. Example for argument "1 1 1 1"

Homework Submission

Task 1 Parallel counter (15%)

Submit four versions of the parallel counter code (mutex, semaphore, pthread_spinlock, and homemade_spinlock). Name the codes as mutex, sem, spinlock, homemade_spinlock. File extensions should be .cpp.

Task 2 Estimate pi with Monte Carlo Method (25%)

Submit two versions of pi estimation code (lock-free and with-lock). There is no restriction on the type of locks (could be spinlock or mutex) that you may use. In addition, please find out how many threads will give the shortest execution time for each version of the code respectively. Name the codes as pi_lock and pi_free. File extensions should be .c or cpp.

Task 3 Cross the intersection (40%)

A program that shows when cars cross the road and the deadlock situation.
Name the code as crossroad.c or crossroad.cpp.

Report (20%)

1. Task1: Compare the execution times with respect to the four synchronizing methods and varying number of threads (use 2, 4, 8, and 16 threads). (10%)
2. Task2: Compare the execution times of the two methods. (5%)
3. Task3: Briefly explain how you detect and solve the deadlock. (5%)
4. Name this report as report.pdf.

Upload a zipped file named as HW2_XXXXXX.zip where XXXXXX is your student id. Homework submission should include seven C/C++ source files which are four synchronizing methods from task1, lock-free and use-lock version from task2, and a deadlock simulation from task3, a report, a Makefile and no folders. As homework1, make sure your program can run on the department workstations linux1~6.cs.nctu.edu.tw, or make sure it is compatible with Ubuntu 14.04 LTS Linux and GCC version 4.8.4 if you don't have access to the department workstations.

P.S. plagiarism (-100%), submission format error (-5%)

--T.A. 黃俊魁(koosan1120@hotmail.com)