Embedded System Software Design

Project 2

Problem Definition:

In this project, we execute the multi-level convolution. By parallelizing some regions in a program, we can reduce the execution duration, but the data dependency might damage the parallelism. In this project, we will protect the shared resource and synchronize threads in multi-thread mode.

Experimental Environment:

- ✓ PC: at least 4 cores
- ✓ RAM: at least 4GB
- ✓ OS: Ubuntu 16.04 or version above
- ✓ Package requirement:
  - o g++ (5.4.0)
  - o GNU make (4.1)

Part1 (Mutex and Barrier):

In part 1, to protect the shared resource by using a mutex, and to synchronize threads by using a barrier in order to obtain the correct result.

a) Describe how to protect share resources and how to synchronize threads in your report.

b) Show the execution result in Figure 2 in your report.

```
======================= System Info =======================
numThread: 4
maskSize: 31 x 31
matrixSize: 1000 x 1000
Protect Shared Resource: Mutex
Synchronize: Barrier

=================== Generate Matrix Data ==================
Generate Date Spend time : 0.009927

============= Start Single Thread Convolution =============
Single Thread Spend time : 18.779

============== Start Multi-Thread Convolution =============
Thread ID : 0   PID : 1812192   Core : 0
Thread ID : 2   PID : 1812194   Core : 2
Thread ID : 1   PID : 1812193   Core : 1
Thread ID : 3   PID : 1812195   Core : 3
Multi Thread Spend time : 9.29079

======================== checking =========================
Error result locate at : [0][0]
6.14614e+06 != 4.7616e+06
Matrix convolution result !!WRONG!!
```

Figure 1 Original program

```
======================= System Info =======================
numThread: 4
maskSize: 31 x 31
matrixSize: 1000 x 1000
Protect Shared Resource: Mutex
Synchronize: Barrier

=================== Generate Matrix Data ==================
Generate Date Spend time : 0.008837

============= Start Single Thread Convolution =============
Single Thread Spend time : 12.0851

============== Start Multi-Thread Convolution =============
Thread ID : 0   PID : 1812139   Core : 0
Thread ID : 1   PID : 1812140   Core : 1
Thread ID : 2   PID : 1812141   Core : 2
Thread ID : 3   PID : 1812142   Core : 3
Multi Thread Spend time : 10.3489

======================== checking =========================
Matrix convolution result correct.
```

Figure 2 Mutex and barrier

Part2 (Reentrant Function):

In part 2, please modify the non-reentrant function into the reentrant function.

a) Describe how to modify the non-reentrant function to the reentrant function in your report.

b) Show the execution result of multi-thread with reentrant function as Figure 3.

c) Analyze the execution time of the non-reentrant function and reentrant function, and compare them. (Please use your experimental result to support your discussion)

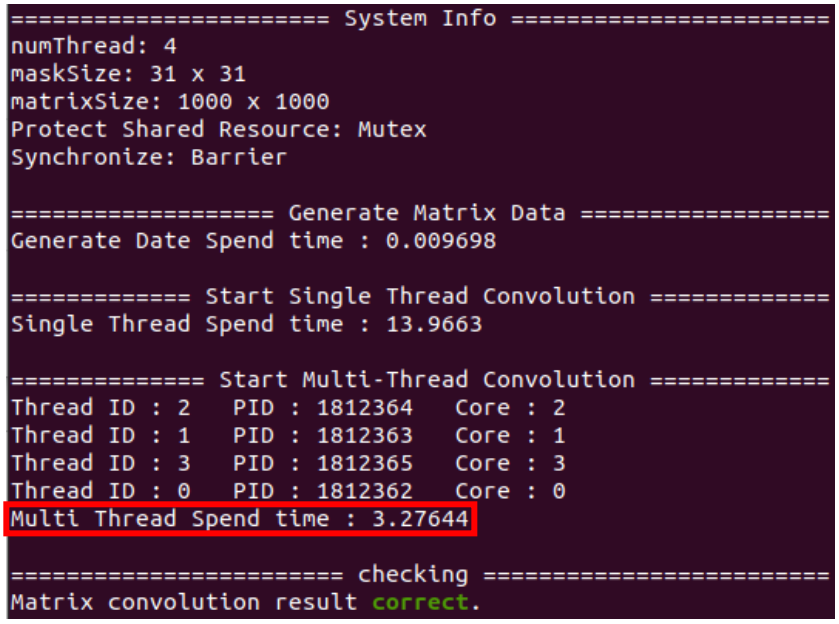


Figure 3 Multi-thread with the reentrant function

Part3 (Spinlock):

In part 3, try to protect the shared resource by using spinlock and synchronize the thread by using a barrier in order to obtain the correct result. Please modify the code based on part 1.

a) Describe how to protect the shared resource by using spinlock.

b) Show the execution result in Figure 4.

c) Compare to part 1, please observe which method could obtain better performance under the benchmark we provided and explain why. Please use the execution results to support your discussion. (Show both the execution results of using mutex and spinlock respectively to support your discussion.)

d) Following (c), please modify the configuration of the benchmark (maskSize and matrixSize) such that the performance results are opposite to the results of (c). Show the configuration of your benchmark by the screenshot of a file (config.h) and describe the property of the configuration. (Show both the execution results of using mutex and spinlock respectively to support your discussion.)

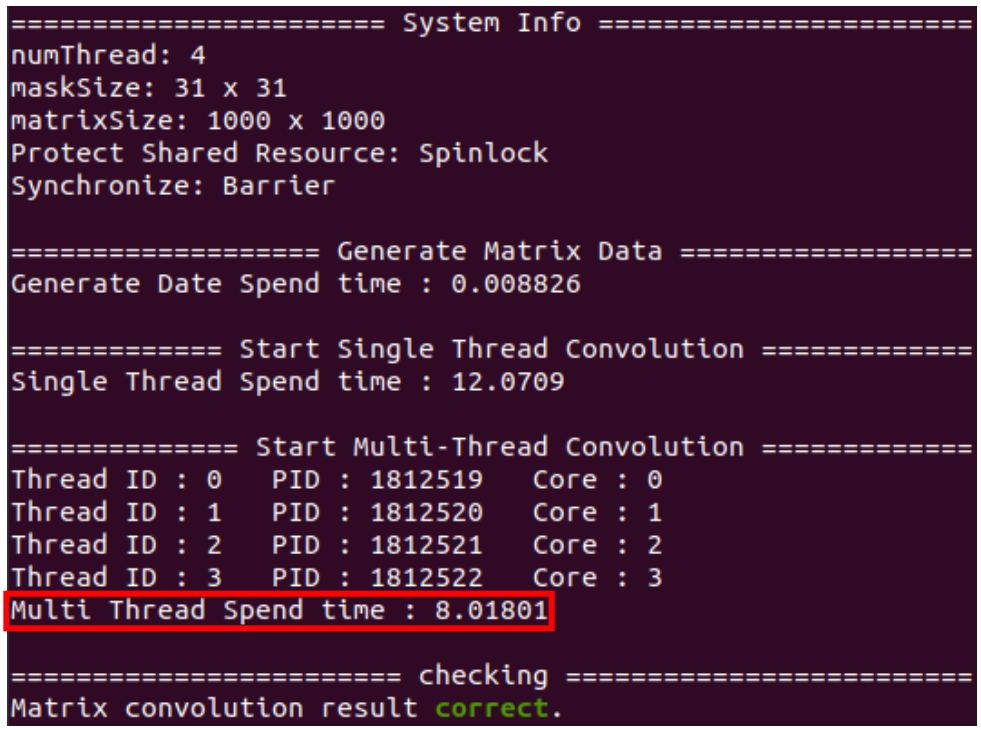


Figure 4 Spinlock and barrier

Command Line:

- Part 1
    - Compile: make part1.out
    - Execute: ./part1.out

- Part 2
    - Compile: make part2.out
    - Execute: ./part2.out

- Part 3
    - Compile: make part3.out
    - Execute: ./part3.out

Crediting:

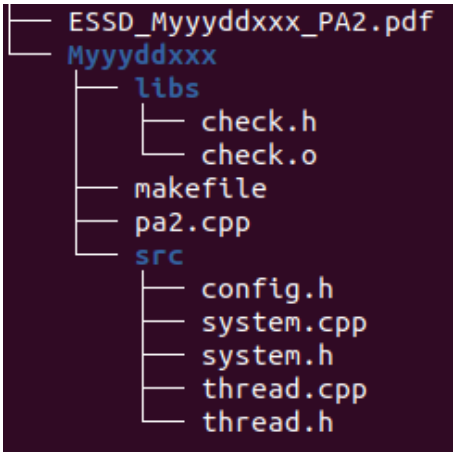
- Part 1 (35%)
    - Execution result of using mutex and barrier. **20%**
    - Describe how to synchronize thread. **10%**
    - Describe how to protect a shared resource. **5%**

- Part 2 (30%)
    - Execution result of using reentrant function. **15%**
    - Describe how to modify non-reentrant function into reentrant function. **10%**
    - Describe the reason why using a non-reentrant function or a reentrant function could obtain better performance. **5%**

- Part 3 (35%)
    - Execution result of using spinlock. **10%**
    - Describe which method (mutex and spinlock) could obtain better performance under the benchmark we provided (**5%**) and why (**5%**).
    - Show the benchmark your used (**5%**), explain the properties of such benchmark (**5%**) and the execution results (**5%**).

Project submits:

- ✓ Submit deadline: 13:00, May, 4, 2022
- ✓ Submission: Moodle
- ✓ Report name: ESSD_Myyyddxxx_PA2.pdf
- ✓ File name format: ESSD_Myyyddxxx_PA2.zip
- ✓ File structure:



- ✓ Note: ESSD_Student ID_PA2.zip must include the **report** (ESSD_Myyyddxxx_PA2.pdf) and **source code** (File name: Myyyddxxx)

嚴禁抄襲，發生該類似情況者，一律以零分計算

Hint:

POSIX Thread Mutex

The mutex object is locked by calling **pthread_mutex_lock ()**. If the mutex is already locked, the calling thread shall be blocked until the mutex becomes available. The **pthread_mutex_unlock ()** function shall release the mutex object referenced by a mutex.

```
#include <pthread.h>

pthread_mutex_t *count_mutex;

pthread_mutex_lock ( count_mutex );
pthread_mutex_unlock ( count_mutex );
```

POSIX Thread SpinLock

The spinlock is a synchronization mechanism suitable primarily for use on multiprocessors with the shared memory.

```
#include <pthread.h>

pthread_spinlock_t *lock;

pthread_spin_init ( lock, pshared );    // pshared is for setting the share flag

pthread_spin_lock ( lock );
pthread_spin_unlock ( lock );
```

Pshared	**Description**
PTHREAD_PROCESS_PRIVATE	Operated only by threads in the same process
PTHREAD_PROCESS_SHARED	Operated by any thread in any process

POSIX Thread Barrier

POSIX threads specify a synchronization object called a barrier, along with barrier functions. The functions create the barrier, specifying the number of threads that are synchronizing on the barrier, and set up threads to perform tasks and wait at the barrier until all the threads reach the barrier. When the last thread arrives at the barrier, all the threads resume execution.

```
#include <pthread.h>

pthread_barrier_t *barr;

pthread_barrier_init ( barr, attr, count );
pthread_barrier_wait ( barr );
```

POSIX Semaphore

POSIX semaphores allow processes and threads to synchronize their actions. A semaphore is an integer whose value is never allowed to fall below zero. Two operations can be performed on semaphores: increment the semaphore value by one (**sem_post()**); and decrement the semaphore value by one (**sem_wait()**). If the value of a semaphore is currently zero, then a **sem_wait()** operation will block until the value becomes greater than zero.

```
#include <semaphore.h>

sem_t *sem;

sem_init ( sem, pshared, value );        // value is for setting counter semaphore

sem_wait ( sem );
sem_post ( sem );
```