Embedded System Software Design

Project 2

Problem Definition:

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 multithread mode.

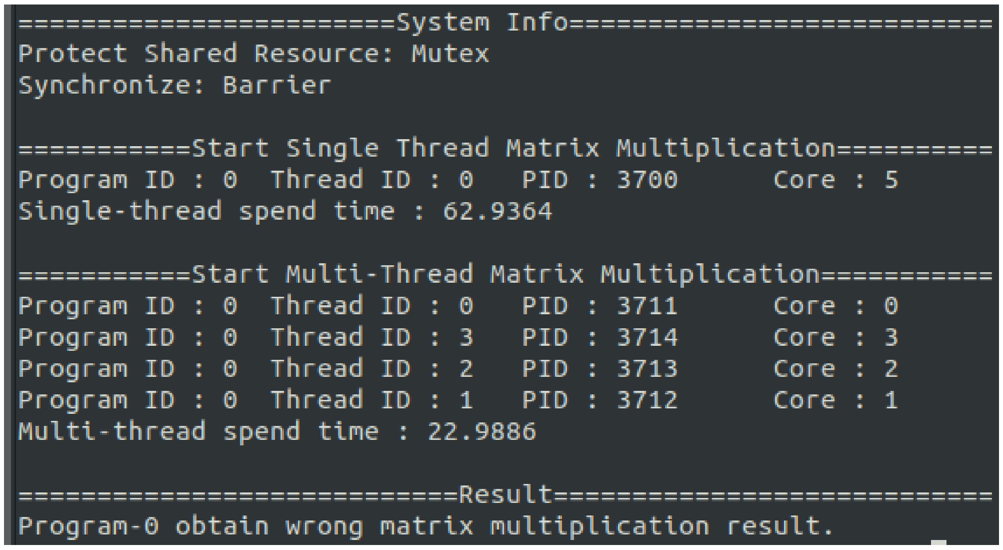
Experimental Environment:

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

Part1 (Mutex and Barrier):

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

a) Describe how to protect share resources and synchronize threads in your report.
b) Show the execution result as Figure 2 in your report.



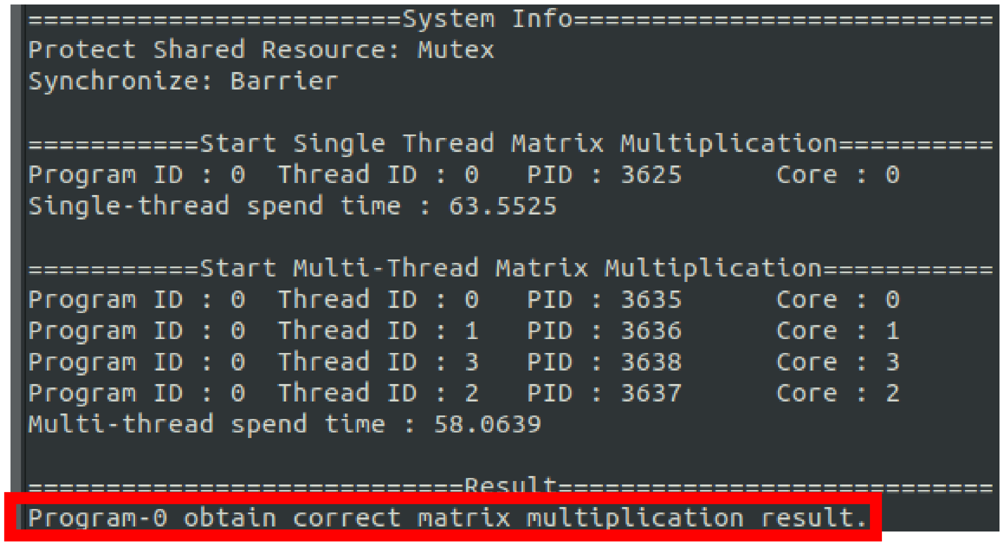
```
========================System Info===========================
Protect Shared Resource: Mutex
Synchronize: Barrier

===========Start Single Thread Matrix Multiplication==========
Program ID : 0  Thread ID : 0   PID : 3700      Core : 5
Single-thread spend time : 62.9364

===========Start Multi-Thread Matrix Multiplication===========
Program ID : 0  Thread ID : 0   PID : 3711      Core : 0
Program ID : 0  Thread ID : 3   PID : 3714      Core : 3
Program ID : 0  Thread ID : 2   PID : 3713      Core : 2
Program ID : 0  Thread ID : 1   PID : 3712      Core : 1
Multi-thread spend time : 22.9886

============================Result============================
Program-0 obtain wrong matrix multiplication result.
```

Figure 1 Original program



```
========================System Info===========================
Protect Shared Resource: Mutex
Synchronize: Barrier

===========Start Single Thread Matrix Multiplication==========
Program ID : 0  Thread ID : 0   PID : 3625      Core : 0
Single-thread spend time : 63.5525

===========Start Multi-Thread Matrix Multiplication===========
Program ID : 0  Thread ID : 0   PID : 3635      Core : 0
Program ID : 0  Thread ID : 1   PID : 3636      Core : 1
Program ID : 0  Thread ID : 3   PID : 3638      Core : 3
Program ID : 0  Thread ID : 2   PID : 3637      Core : 2
Multi-thread spend time : 58.0639

============================Result============================
Program-0 obtain correct matrix multiplication result.
```

Figure 2 Mutex and barrier

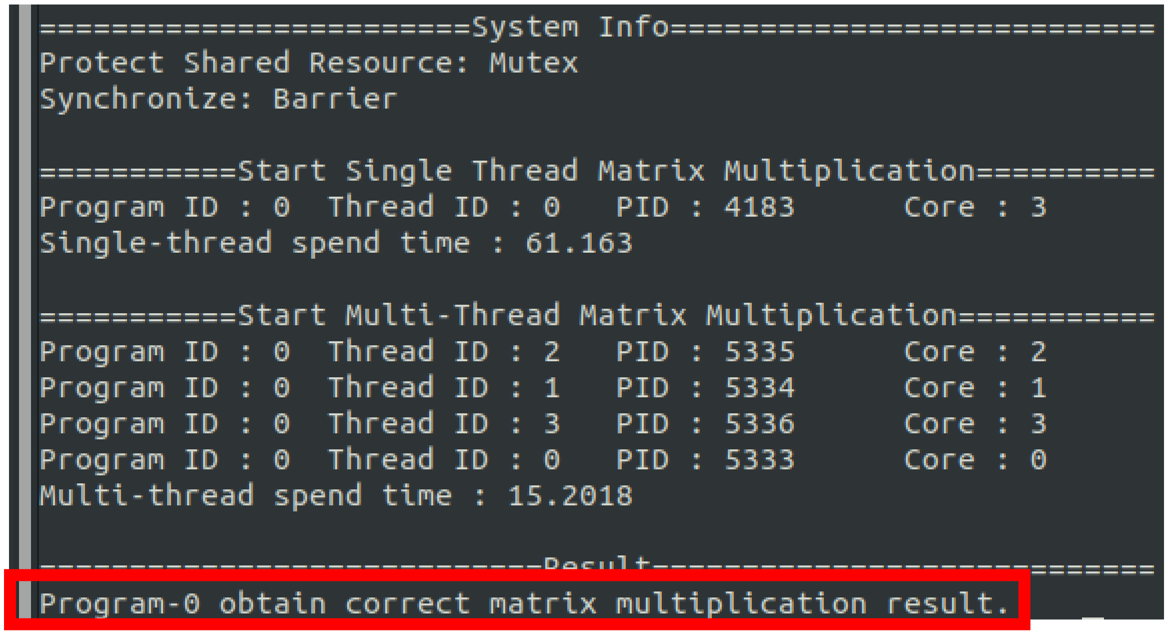
Part2 (reentrant Function):

In part 2, please modify the non-reentrant function to 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)



```
========================System Info============================
Protect Shared Resource: Mutex
Synchronize: Barrier

===========Start Single Thread Matrix Multiplication==========
Program ID : 0  Thread ID : 0   PID : 4183      Core : 3
Single-thread spend time : 61.163

===========Start Multi-Thread Matrix Multiplication===========
Program ID : 0  Thread ID : 2   PID : 5335      Core : 2
Program ID : 0  Thread ID : 1   PID : 5334      Core : 1
Program ID : 0  Thread ID : 3   PID : 5336      Core : 3
Program ID : 0  Thread ID : 0   PID : 5333      Core : 0
Multi-thread spend time : 15.2018

===========================Result=============================
Program-0 obtain correct matrix multiplication result.
```

Figure 3 Multi-thread with the reentrant function

Part3 (Spinlock):

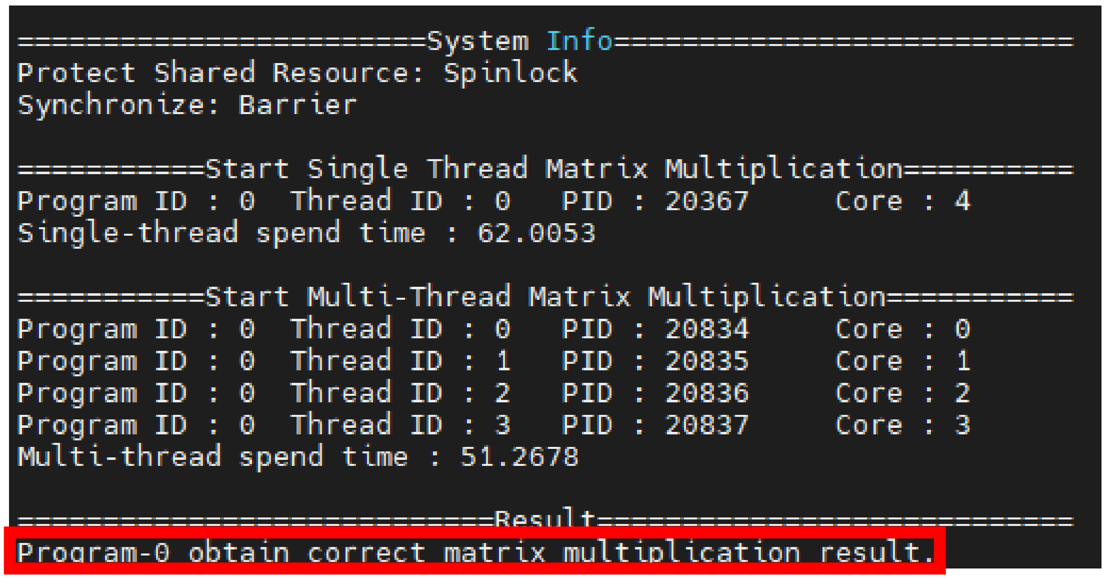
In part 3, try to protect the shared resource by using spinlock and synchronize thread by using barrier in ordered to obtain the correct result. Please modify the code based on part 1.

a) Describe how to protect shared resource.

b) Show the execution result as Figure 4.

c) Compare to part1, 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 find a configuration of the benchmark 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.)



```
========================System Info===========================
Protect Shared Resource: Spinlock
Synchronize: Barrier

===========Start Single Thread Matrix Multiplication==========
Program ID : 0  Thread ID : 0   PID : 20367     Core : 4
Single-thread spend time : 62.0053

===========Start Multi-Thread Matrix Multiplication===========
Program ID : 0  Thread ID : 0   PID : 20834     Core : 0
Program ID : 0  Thread ID : 1   PID : 20835     Core : 1
Program ID : 0  Thread ID : 2   PID : 20836     Core : 2
Program ID : 0  Thread ID : 3   PID : 20837     Core : 3
Multi-thread spend time : 51.2678

============================Result============================
Program-0 obtain correct matrix multiplication result.
```

Figure 4 Spinlock and barrier

<u>Command Line:</u>

- 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

<u>Crediting:</u>

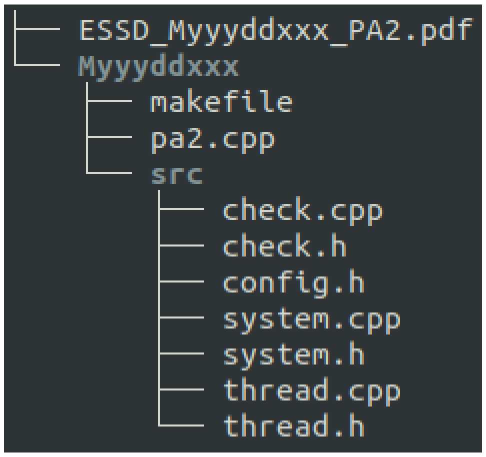
- Part 1 (35%)
    - Execution result of using mutex and barrier. **20%**
    - Describe how to synchronize thread. 1**0%**
    - Describe how to protect a shared resource. 5**%**

- 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: 12:30, May. 26, 2021
- ✓ Submission: Moodle
- ✓ Report name: ESSD_Myyyddxxx_PA2.pdf
- ✓ File name format: ESSD_Myyyddxxx_PA2.zip
- ✓ File structure:



```
├── ESSD_Myyyddxxx_PA2.pdf
└── Myyyddxxx
    ├── makefile
    ├── pa2.cpp
    └── src
        ├── check.cpp
        ├── check.h
        ├── config.h
        ├── system.cpp
        ├── system.h
        ├── thread.cpp
        └── thread.h
```

- ✓ 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);
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);
sem_wait (&sem);
sem_post (&sem);
```