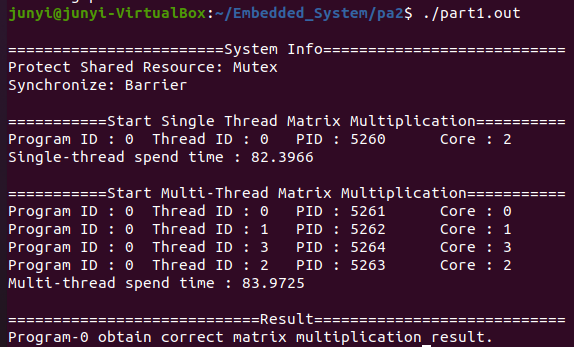
**Embedded System Software Design Project 2**

M10907324吳俊逸

***• Part 1 (35%)***

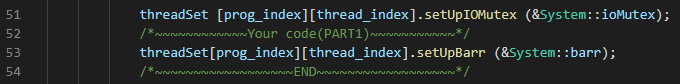
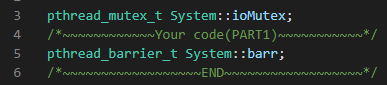
***o Execution result of using mutex and barrier. 20%***



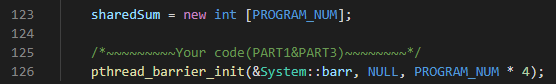
***o Describe how to synchronize thread. 10%***

***in system.cpp：***

跟著ioMutex設定Barrier參數

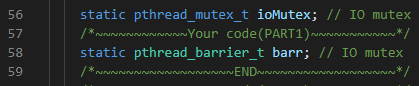
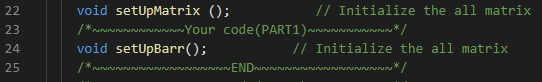


初始化Barrier以及設定等待個數為4倍的PROGRAM\_NUM，因為我們使用4個core，所以我必須設定他一定要等待4個core全部到齊才可以執行下去。



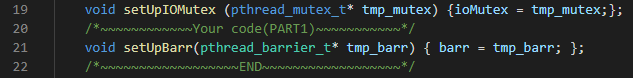
***in system.h：***

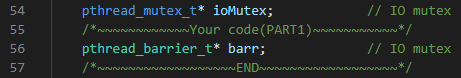
跟著ioMutex設定Barrier參數



***In threat.h：***

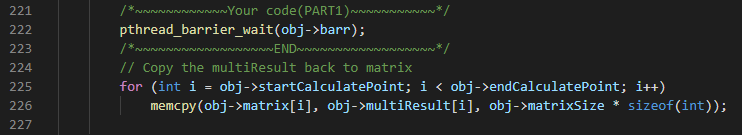
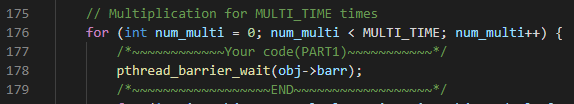
跟著ioMutex設定Barrier參數

******

******

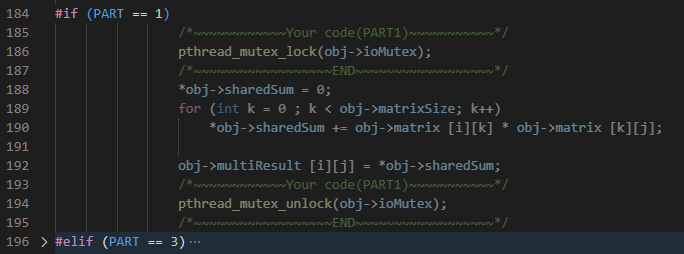
***In threat.cpp：***

首先我先在每一個core進來的地方放一個Barrier的等待指令(175~179行)，讓所有的core可以一起抵達之後再執行，最後在要存進memory之前再放一個Barrier的等待指令(221~227)，讓所有core在同一時間開始將資料儲存至memory之中。避免運算好的資料被覆蓋，導致結果出錯。

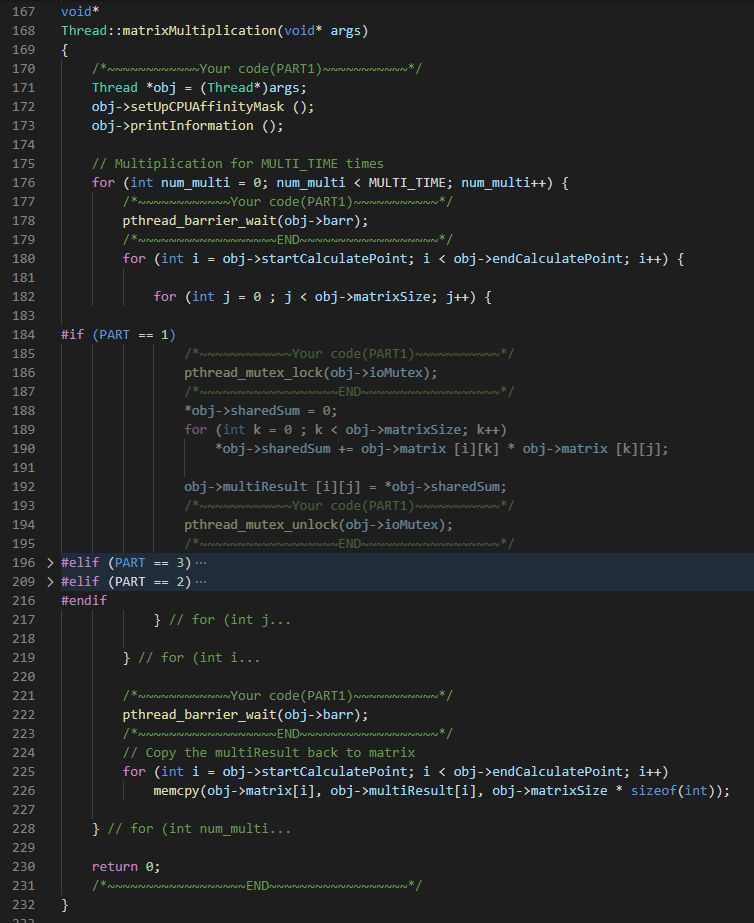


***o Describe how to protect a shared resource. 5%***

我在有用到共用資源的地方也就是矩陣相乘的地方(188~192行)前後利用mutex將資源鎖住(186、194行)，不讓其他core使用

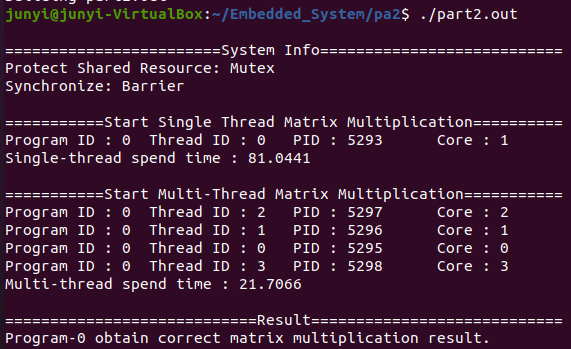


***In threat.cpp完整的code：***



***• Part 2 (30%)***

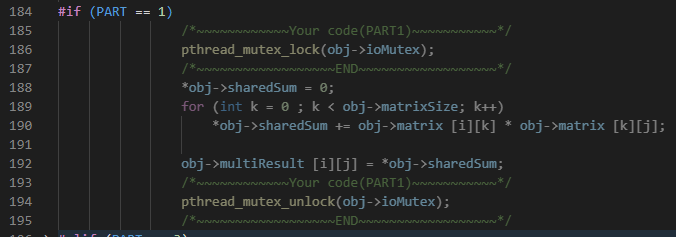
***o Execution result of using reentrant function. 15%***



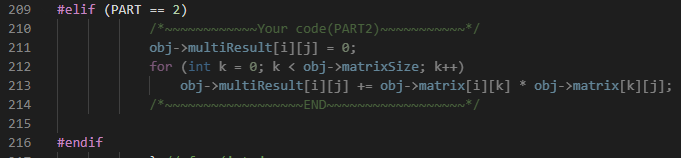
***o Describe how to modify non-reentrant function into reentrant function. 10%***

***In threat.cpp：***

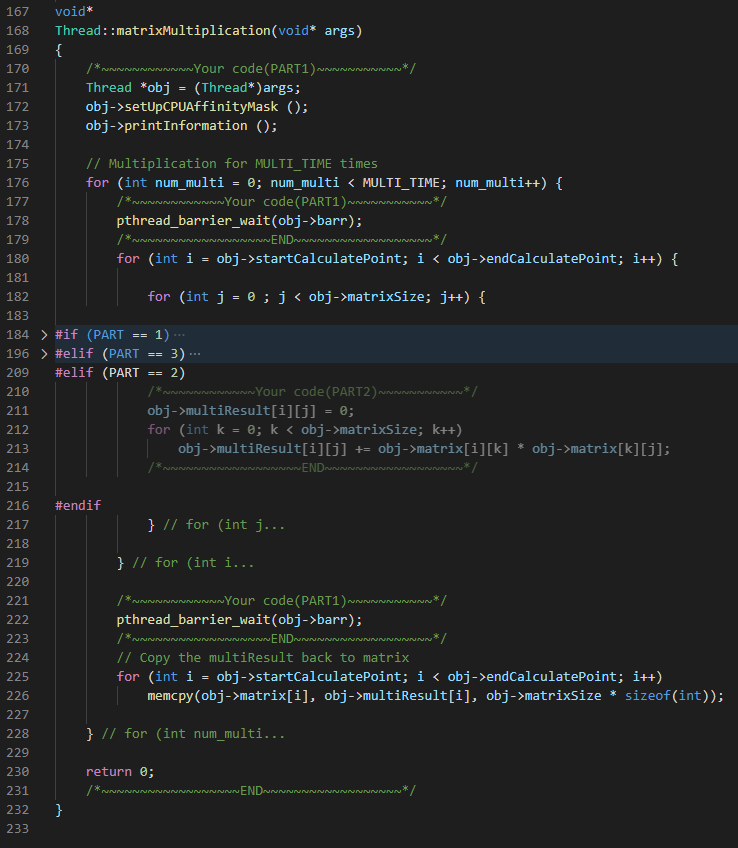
非重入函數是不能由多個任務共享的函數，需要保護資源也是就先把運算結果放在sharesum，算完再放到multiresult(186~194行)。



相反，可重入函數是可以同時由多個任務使用的函數，而不必擔心數據損壞，可以直接放到multiresult中(210~214行)。



***In threat.cpp完整的code：***



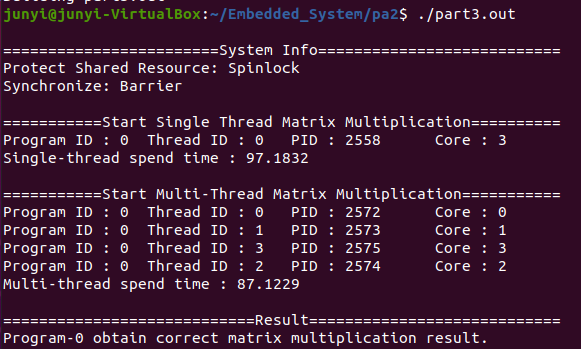
***o Describe the reason why using a non-reentrant function or a reentrant function could obtain better performance. 5%***

可重入（reentrant）函式可以由多於一個任務並發使用，而不必擔心數據錯誤。相反，不可重入（non-reentrant）函式不能由超過一個任務所共享，除非能確保函式的互斥（或者使用信號量，或者在程式碼的關鍵部分禁用中斷）。也就是說reentrant可以做到資源同時給所有的core去執行，可以完美的進行併行化處理，反觀non-reentrant資源只能給一個core使用，就像是一個single processor一樣，我們也可以從part1、part2的實驗結果得知part1的Multi-Thread執行時間與Single Thread幾乎一樣，還比Single Thread花費較多的時間；part2的Multi-Thread執行時間與Single Thread相差近四倍。由此可知reentrant function比non-reentrant function還要好。

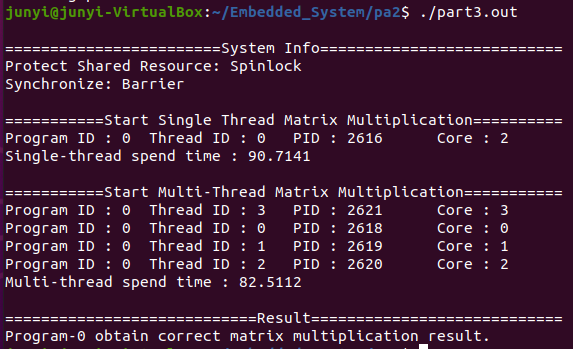
***• Part 3 (35%)***

***o Execution result of using spinlock. 10%***

1. ***use PRIVATE***



1. ***use SHARED***

******

***o Describe which method (mutex and spinlock) could obtain better performance under the benchmark we provided (5%) and why (5%).***

在單核心中：mutex比較好

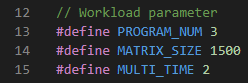
在多核心中：spinlock比較好

WHY：

mutex跟spinlock來比較，mutex的機制是當process無法鎖定mutex時，process會進入sleep，中間會需要付出context switch的代價。spinlock的機制是採用 busy-waiting，直到鎖定spinlock為止。依據兩者的特性，一般來說，在單核心的系統中，會採用mutex，因為spinlock的busy waiting沒意義（沒有CPU可以釋放鎖）。而多核心，且critical section只耗用一小段時間的狀況下，適合使用spinlock。

***o Show the benchmark your used (5%), explain the properties of such benchmark (5%) and the execution results (5%).***

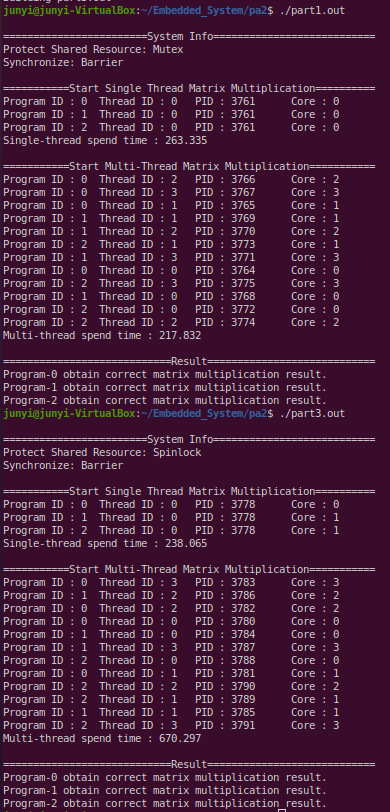
Benchmark：



Explain：

因為spinlock是一直等待指定的鎖被釋放之後，才可以繼續進行下一步動作；然而mutex在等待的時候會先讓其他不需要用資源的先做。這也就代表著只要不斷的增加PROGRAM數量，spinlock的總共等待時間一定會增加的比mutex還要來的多，最後就會達成mutex比spinlock還要快的情形發生。

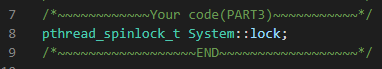
Resual：



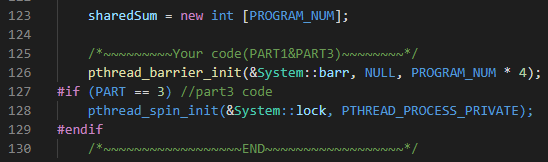
***Describe how to protect a shared resource. (use spinlock)***

***in system.cpp：***

跟著ioMutex設定Spinlock參數

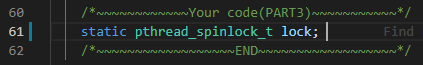
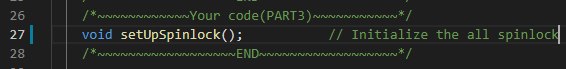


設定spinlock初始化



***in system.h：***

跟著ioMutex設定Spinlock參數



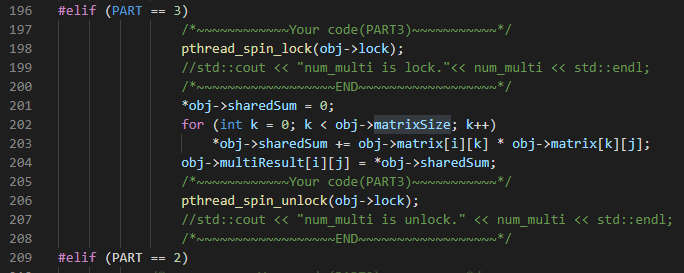
***In threat.h：***

跟著ioMutex設定Spinlock參數



***In threat.cpp：***

我在有用到共用資源的地方也就是矩陣相乘的地方(200~205行)前後利用spinlock將資源鎖住(198、206行)，不讓其他core使用



***In threat.cpp完整的code：***

