嵌入式系統-PA1

學號：M11007326 姓名：黃鈞臨

- **Part 1**

[Global Scheduling 10%]

- ■ **Describe how to implement Global scheduling by using pthread. 5%**

```
System::globalMultiCoreConv ()
{
    std::cout << "\n===========Start Global Multi-Thread Convolution===========" << std::endl;
    check->setCheckState(GLOBAL);
    setStartTime();

    /*~~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~*/
    // Create thread and join

    for(int i=0;i<numThread;i++){
        pthread_create (&threadSet[i]._thread, NULL, threadSet[i].convolution, &threadSet[i]);
    }
    for(int i=0;i<numThread;i++){
        pthread_join (threadSet[i]._thread, NULL);
    }

    /*~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~*/

    setEndTime();
    std::cout << "Global Multi Thread Spend time : " << _timeUse << std::endl;
    cleanMultiResult();
}
```

用 Pthread_create 建立新的執行緒，建立好的執行緒會以平行的方式執行，而最後再用 Pthread_join 函數等待執行緒結束，而在設置 Global scheduling 時不必使用 setCore 設定 cpu，因為 setCore 的預設值是-1，而在-1 的情況下 thread 會配給到隨機的 cpu 上。

- ■ **Describe how to observe task migration. 5%**

```
#if (PART == 1)
    pthread_mutex_lock(&count_Mutex);
    now = sched_getcpu();
    if(now!= pre){
        now = sched_getcpu();
        std::cout << "The thread " << obj->_ID << " PID : " << obj->PID << " is moved from CPU" <<pre <<" to CPU" << now << std::endl;
        pre = now;
     }
    pthread_mutex_unlock(&count_Mutex);
#endif
```

為了觀察 CPU 的變化，在進到 convolution 的 for 迴圈前，令兩個變數 now 跟 pre 並把當前的 core 丟進去，之後在 for 迴圈裡檢查 cpu 是否有變更，有的話就 cout 出來並更新 pre。

[Partition Scheduling. 5%]

- ■ **Describe how to implement partition scheduling by using pthread**

```
void
System::partitionMultiCoreConv ()
{
#if (PART == 1)
    std::cout << "\n===========Start Partition Multi-Thread Convolution===========" << std::endl;
    check->setCheckState(PARTITION);
    setStartTime();
    for(int i=0;i<numThread;i++){
        threadSet[i].setCore(i%CORE_NUM);
    }
```

```cpp
for(int i=0;i<numThread;i++){
    pthread_create (&threadSet[i]._thread, NULL, threadSet[i].convolution, &threadSet[i]);
}
for(int i=0;i<numThread;i++){
    pthread_join (threadSet[i]._thread, NULL);
}
```

```cpp
void
Thread::setUpCPUAffinityMask (int core_num)
{
    /*~~~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~~*/
   // Pined the thread to core.
    if(core_num == -1){
        return;
     }
    cpu_set_t set;
    CPU_ZERO(&set);
    CPU_SET(core_num,&set);
    if(sched_setaffinity(0,sizeof(set),&set)==-1)
        std::cout << "WRONG ";
    cur_core = core_num;

//  setThreadCore=-1;
    /*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~*/
}
```

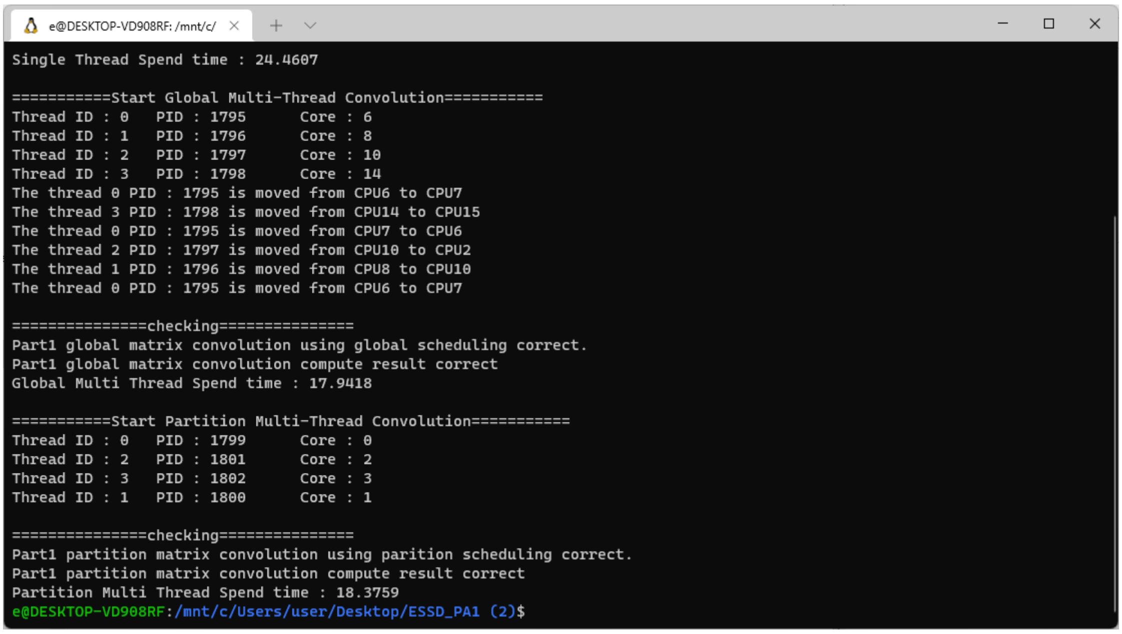
在 partition 時，跟 global 的差異在於，必須將 thread 分配給指定的 cpu，因此這邊使用 setCore 將任務分配到指定的 core 上，之後在利用 setUpCPUAffinityMask 函式設置 CPU。

[Result. 10%]

- **Show the scheduling states of tasks. (You have to show the screenshot result of using the input part1_input.txt)**

```
e@DESKTOP-VD908RF: /mnt/c/
e@DESKTOP-VD908RF:/mnt/c/Users/user/Desktop/ESSD_PA1 (2)$ ./part1.out ./input/part1_Input.txt
Input File Name : ./input/part1_Input.txt
numThread : 4
0.Matrix size : 12000
1.Matrix size : 12000
2.Matrix size : 12000
3.Matrix size : 12000

===========Generate Matrix Data===========
Generate Date Spend time : 5.202

===========Start Single Thread Convolution==========
Single Thread Spend time : 24.4607

===========Start Global Multi-Thread Convolution==========
Thread ID : 0   PID : 1795      Core : 6
Thread ID : 1   PID : 1796      Core : 8
Thread ID : 2   PID : 1797      Core : 10
Thread ID : 3   PID : 1798      Core : 14
The thread 0 PID : 1795 is moved from CPU6 to CPU7
The thread 3 PID : 1798 is moved from CPU14 to CPU15
The thread 0 PID : 1795 is moved from CPU7 to CPU6
The thread 2 PID : 1797 is moved from CPU10 to CPU2
The thread 1 PID : 1796 is moved from CPU8 to CPU10
The thread 0 PID : 1795 is moved from CPU6 to CPU7

==============checking==============
Part1 global matrix convolution using global scheduling correct.
Part1 global matrix convolution compute result correct
Global Multi Thread Spend time : 17.9418
```

▲Global Scheduling



▲Partition Scheduling

- **Part 2**

[Partition method Implementation. 10%]

- **Describe how to implement the three different partition methods (First-Fit, Best-Fit, Worst-Fit) in partition scheduling.**

```
void
System::partitionFirstFit ()
{
    std::cout << "\n===========Partition First-Fit Multi Thread Matrix Multiplication===========" << std::endl;
#if (PART == 2)
    check->setCheckState(PARTITION_FF);
#endif
    for (int i = 0; i < CORE_NUM; i++)
        cpuSet[i].emptyCPU();

    /*~~~~~~~~~~~~Your code(PART2)~~~~~~~~~~~*/
    // Implement parititon first-fit and print result.
    for(int i=0;i<numThread;i++){
        int q=0;

        while(cpuSet[q].utilization()+threadSet[i].utilization()>1){
            q = q+1;
            if(q>CORE_NUM){
                threadSet[i].setCore(-1);
                std::cout<<"Thread-"<<threadSet[i].ID() << " not schedulable"<<std::endl;
                continue;
            }
        }
        if(q<CORE_NUM){
//          std::cout << q<<"_utilization : " << cpuSet[q].utilization() << std::endl;
            cpuSet[q].push_thread(threadSet[i].ID(),threadSet[i].utilization());
//          std::cout << q<<"_afterutilization : " << cpuSet[q].utilization() << std::endl;
            threadSet[i].setCore(q);
        }

    }
    // ...
    // ...
    /*~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~*/

    partitionMultiCoreConv();
    cleanMultiResult();
}
```

First-Fit 的排法要先將前面的 cpu 排滿後才排下一顆，因此這邊用一個 while 判斷是否排滿，排滿的話就排下一顆，而排到超過指定的 CPU 數量後，就判斷這個任務無法排程，並把 core 設成-1。

```cpp
201 void
202 System::partitionBestFit ()
203 {
204     std::cout << "\n===========Partition Best-Fit Multi Thread Matrix Multiplication===========" << std::endl;
205 #if (PART == 2)
206     check->setCheckState(PARTITION_BF);
207 #endif
208 
209     /*~~~~~~~~~~~~Your code(PART2)~~~~~~~~~~~*/
210     // Implement partition best-fit and print result.
211     for (int i = 0; i < CORE_NUM; i++)
212         cpuSet[i].emptyCPU(); // Reset the CPU set
213     for(int i=0;i<numThread;i++){
214         int q=-1;
215         float maxcutilization = -1;
216         for(int j=0;j<CORE_NUM;j++){
217             if(cpuSet[j].utilization()+threadSet[i].utilization()<=1){
218                 if(maxcutilization<cpuSet[j].utilization()){
219                     maxcutilization = cpuSet[j].utilization();
220                     q=j;
221                 }
222             }
223             else{
224                 if(j==(CORE_NUM-1)&&q==-1){
225                     std::cout<<"Thread-"<< i << " not schedulable"<<std::endl;
226                     threadSet[i].setCore(-1);
227                 }
228                 continue;
229             }
230         }
231         if(q!=-1){
232             cpuSet[q].push_thread(threadSet[i].ID(),threadSet[i].utilization());
233             threadSet[i].setCore(q);
234         }
235 
236     }
237     /*~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~*/
238 
239     partitionMultiCoreConv();
240     cleanMultiResult();
241 }
```

Best-Fit 的排法要先把當前利用率最高的先排滿，因此必須在排每個任務時都去比較每個 cpu 的使用率，上圖第 216 行的迴圈就是在比較 cpu 的使用率，而當 cpu 的超過指定的數量時，判斷任務無法排成，並將 core 設成-1。

```cpp
void
System::partitionWorstFit ()
{
    std::cout << "\n===========Partition Worst-Fit Multi Thread Matrix Multiplication===========" << std::endl;
#if (PART == 2)
    check->setCheckState(PARTITION_WF);
#endif

    /*~~~~~~~~~~~~Your code(PART2)~~~~~~~~~~~*/
    // Implement partition worst-fit and print result.
    for (int i = 0; i < CORE_NUM; i++)
        cpuSet[i].emptyCPU(); // Reset the CPU set
    for(int i=0;i<numThread;i++){
        int q=-1;
        float mincutilization = 1;
        for(int j=0;j<CORE_NUM;j++){
            if(cpuSet[j].utilization()+threadSet[i].utilization()<=1){
                if(mincutilization>cpuSet[j].utilization()){
                    mincutilization = cpuSet[j].utilization();
                    q=j;

                }
            }
            else{
                if(j==3&&q==-1){
                    std::cout<<"Thread-"<< i << " not schedulable"<<std::endl;
                    threadSet[i].setCore(-1);
                }
                continue;
            }
        }
        if(q!=-1){
            cpuSet[q].push_thread(threadSet[i].ID(),threadSet[i].utilization());
            threadSet[i].setCore(q);
        }

    }
    /*~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~*/
    partitionMultiCoreConv();
    cleanMultiResult();
}
```

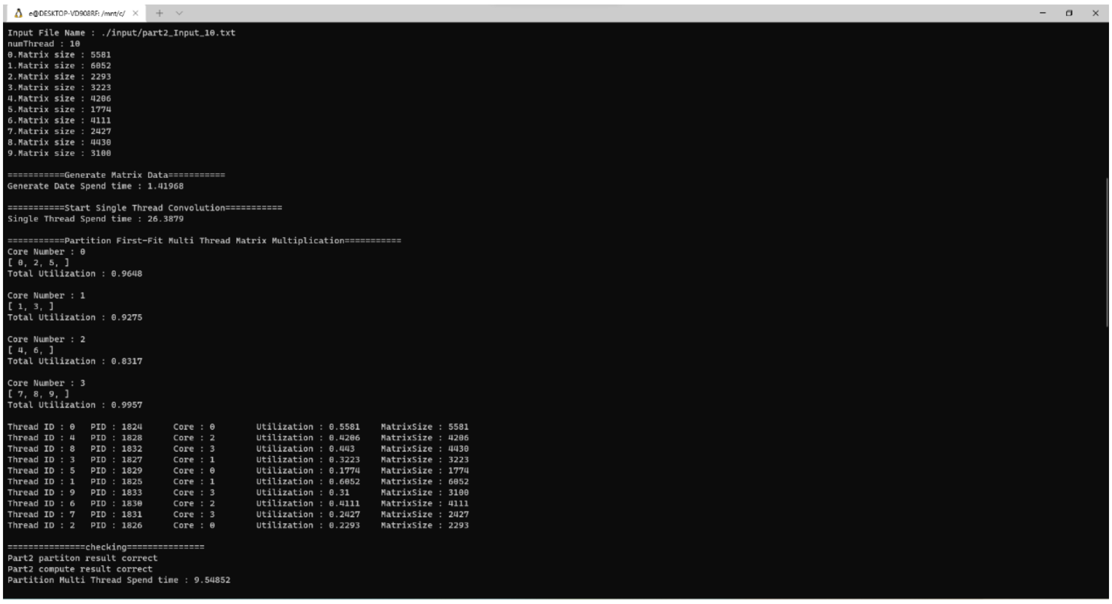
Worst Fit 的排法是先選擇利用率最小的 cpu 排程，與 Best Fit 相反，在程式上也差不多，只是將原本找出最大的 cpu 改成找出最小的 cpu，其他就都一樣。

[Result. 30%]

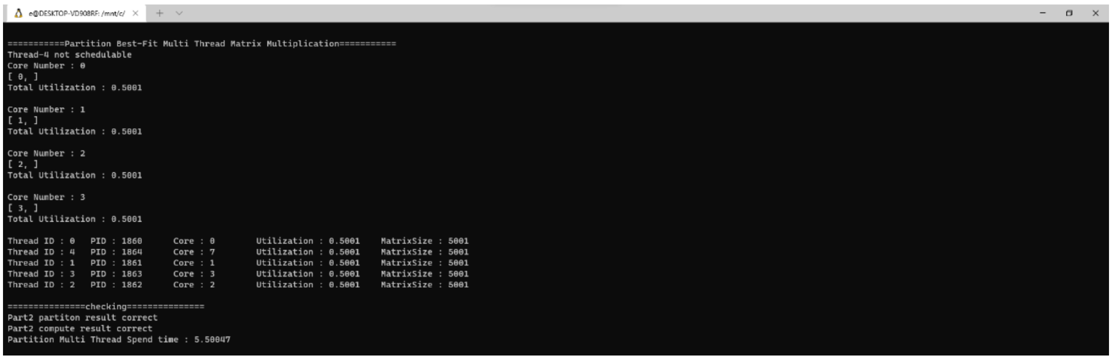
- **Show the scheduling states of tasks. (You have to show the screenshot result of using input part2_input_5.txt and part2_input_10.txt)**

```
e@DESKTOP-VD908RF:/mnt/c/Users/user/Desktop/ESSD_PA1 (2)$ ./part2.out ./input/part2_Input_5.txt
Input File Name : ./input/part2_Input_5.txt
numThread : 5
0.Matrix size : 5001
1.Matrix size : 5001
2.Matrix size : 5001
3.Matrix size : 5001
4.Matrix size : 5001

===========Generate Matrix Data===========
Generate Date Spend time : 1.09332

===========Start Single Thread Convolution===========
Single Thread Spend time : 20.8245

===========Partition First-Fit Multi Thread Matrix Multiplication===========
Core Number : 0
[ 0, ]
Total Utilization : 0.5001

Core Number : 1
[ 1, ]
Total Utilization : 0.5001

Core Number : 2
[ 2, ]
Total Utilization : 0.5001

Core Number : 3
[ 3, ]
Total Utilization : 0.5001

Thread ID : 0   PID : 1855    Core : 0     Utilization : 0.5001    MatrixSize : 5001
Thread ID : 1   PID : 1856    Core : 1     Utilization : 0.5001    MatrixSize : 5001
Thread ID : 4   PID : 1859    Core : 14    Utilization : 0.5001    MatrixSize : 5001
Thread ID : 2   PID : 1857    Core : 2     Utilization : 0.5001    MatrixSize : 5001
Thread ID : 3   PID : 1858    Core : 3     Utilization : 0.5001    MatrixSize : 5001

===============checking===============
Part2 partiton result correct
Part2 compute result correct
Partition Multi Thread Spend time : 5.47083
```

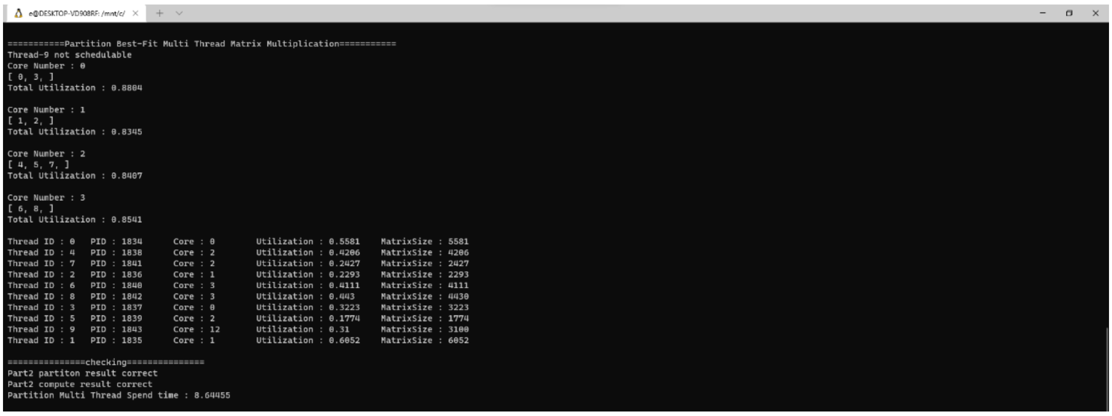
▲First-Fit with input_5.txt



▲First-Fit with input_10.txt



▲Best-Fit with input_5.txt



▲Best-Fit with input_10.txt

```
===========Partition Worst-Fit Multi Thread Matrix Multiplication===========
Thread-4 not schedulable
Core Number : 0
[ 0, ]
Total Utilization : 0.5001

Core Number : 1
[ 1, ]
Total Utilization : 0.5001

Core Number : 2
[ 2, ]
Total Utilization : 0.5001

Core Number : 3
[ 3, ]
Total Utilization : 0.5001

Thread ID : 0   PID : 3091     Core : 0       Utilization : 0.5001    MatrixSize : 5001
Thread ID : 1   PID : 3092     Core : 1       Utilization : 0.5001    MatrixSize : 5001
Thread ID : 2   PID : 3093     Core : 2       Utilization : 0.5001    MatrixSize : 5001
Thread ID : 3   PID : 3094     Core : 3       Utilization : 0.5001    MatrixSize : 5001
Thread ID : 4   PID : 3095     Core : 13      Utilization : 0.5001    MatrixSize : 5001

===============checking===============
Part2 partiton result correct
Part2 compute result correct
Partition Multi Thread Spend time : 5.67225
```

▲Worst-Fit with input_5.txt

```
===========Partition Worst-Fit Multi Thread Matrix Multiplication===========
Thread-8 not schedulable
Core Number : 0
[ 0, 7, ]
Total Utilization : 0.8008

Core Number : 1
[ 1, 9, ]
Total Utilization : 0.9152

Core Number : 2
[ 2, 4, ]
Total Utilization : 0.6499

Core Number : 3
[ 3, 5, 6, ]
Total Utilization : 0.9108

Thread ID : 0   PID : 1844     Core : 0       Utilization : 0.5581    MatrixSize : 5581
Thread ID : 5   PID : 1849     Core : 3       Utilization : 0.1774    MatrixSize : 1774
Thread ID : 3   PID : 1847     Core : 3       Utilization : 0.3223    MatrixSize : 3223
Thread ID : 1   PID : 1845     Core : 1       Utilization : 0.6052    MatrixSize : 6052
Thread ID : 2   PID : 1846     Core : 2       Utilization : 0.2293    MatrixSize : 2293
Thread ID : 8   PID : 1852     Core : 8       Utilization : 0.443     MatrixSize : 4430
Thread ID : 6   PID : 1850     Core : 3       Utilization : 0.4111    MatrixSize : 4111
Thread ID : 9   PID : 1853     Core : 1       Utilization : 0.31      MatrixSize : 3100
Thread ID : 4   PID : 1848     Core : 2       Utilization : 0.4206    MatrixSize : 4206
Thread ID : 7   PID : 1851     Core : 0       Utilization : 0.2427    MatrixSize : 2427

===============checking===============
Part2 partiton result correct
Part2 compute result correct
Partition Multi Thread Spend time : 8.59189
e@DESKTOP-VD908RF:/mnt/c/Users/user/Desktop/ESSD_PA1 (2)$
```

▲Worst-Fit with input_10.txt

- **Part 3**

  [Scheduler Implementation. 10%]

  - **Describe how to implement the scheduler setting in partition scheduling. (FIFO with FF, RR with FF)**

```
#if (PART == 3)
        /*~~~~~~~~~~~~~~Your code(PART3)~~~~~~~~~~~~*/
        // Set the scheduling policy for thread.

        threadSet[i].setSchedulingPolicy(SCHEDULING);
        // ...
        // ...
        /*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~*/
#endif
```

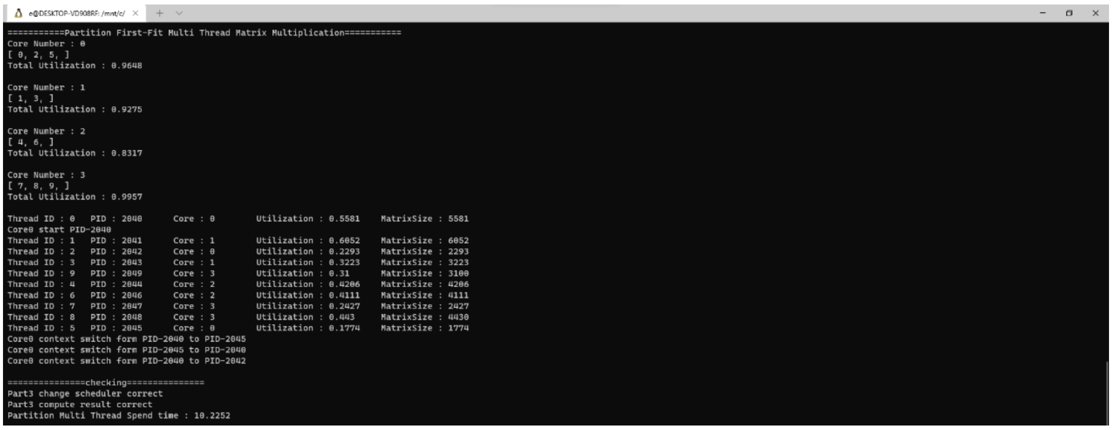
首先在 system.cpp 中根據讀進來的 SCHEDULING 設定 FIFO 或是 Round-Robin

```
void
Thread::setUpScheduler()
{
    /*~~~~~~~~~~~~~~Your code(PART3)~~~~~~~~~~~~*/
  // Set up the scheduler for current thread
    struct sched_param param;
    int maxpri;
    maxpri = sched_get_priority_max(_schedulingPolicy);
    // std::cout << "setUpScheduler" <<std::endl;
    if(maxpri == -1){
        std::cout << "!! Failed sched_get_priority_max_ !!" << std::endl;
    }
    param.sched_priority = maxpri;
    PID = syscall (SYS_gettid);
    int ret = sched_setscheduler (PID, _schedulingPolicy, &param);
    if (ret ==-1) {
        std::cout << "!! Failed sched_setscheduler_ !!" << std::endl;
    }
    /*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~*/
}
```

接著在 setUpScheduler 裡的 sched_get_priority_max()拿到該排程的最大優先權，再透過 sched_setscheduler (PID, _schedulingPolicy, &param);設置排程就完成了。

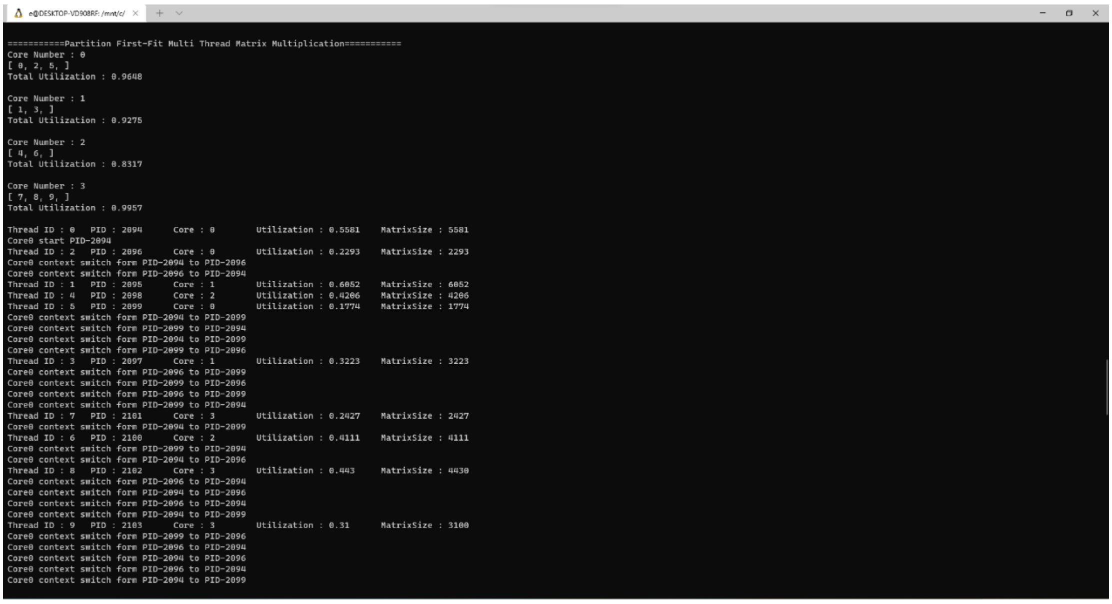
[Result. 10%]

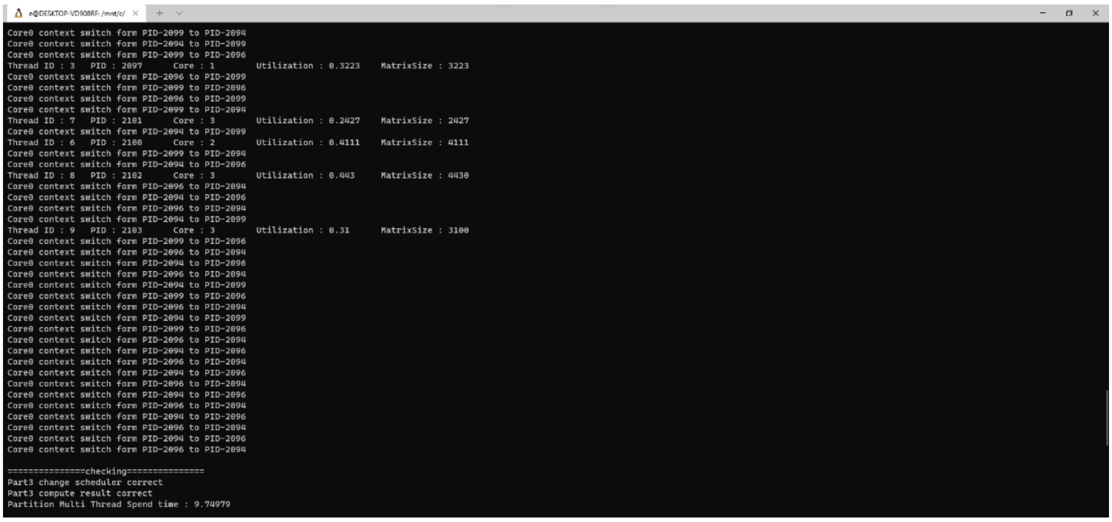
- **Show the process execution states of tasks. (You have to show the screen shot result of using input part3_input.txt)**



```
e@DESKTOP-VD908RF: /mnt/c/
===========Partition First-Fit Multi Thread Matrix Multiplication===========
Core Number : 0
[ 0, 2, 5, ]
Total Utilization : 0.9648

Core Number : 1
[ 1, 3, ]
Total Utilization : 0.9275

Core Number : 2
[ 4, 6, ]
Total Utilization : 0.8317

Core Number : 3
[ 7, 8, 9, ]
Total Utilization : 0.9957

Thread ID : 0   PID : 2040      Core : 0        Utilization : 0.5581    MatrixSize : 5581
Core0 start PID-2040
Thread ID : 1   PID : 2041      Core : 1        Utilization : 0.6052    MatrixSize : 6052
Thread ID : 2   PID : 2042      Core : 0        Utilization : 0.2293    MatrixSize : 2293
Thread ID : 3   PID : 2043      Core : 1        Utilization : 0.3223    MatrixSize : 3223
Thread ID : 9   PID : 2049      Core : 3        Utilization : 0.31      MatrixSize : 3100
Thread ID : 4   PID : 2044      Core : 2        Utilization : 0.4206    MatrixSize : 4206
Thread ID : 6   PID : 2046      Core : 2        Utilization : 0.4111    MatrixSize : 4111
Thread ID : 7   PID : 2047      Core : 3        Utilization : 0.2427    MatrixSize : 2427
Thread ID : 8   PID : 2048      Core : 3        Utilization : 0.443     MatrixSize : 4430
Thread ID : 5   PID : 2045      Core : 0        Utilization : 0.1774    MatrixSize : 1774
Core0 context switch form PID-2040 to PID-2045
Core0 context switch form PID-2045 to PID-2040
Core0 context switch form PID-2040 to PID-2042

===============checking===============
Part3 change scheduler correct
Part3 compute result correct
Partition Multi Thread Spend time : 10.2252
```

▲FIFO





▲Round-Robin

- **Discussion**
  - **Analyze and compare the response time of the program, with single thread and multi-thread using in part 1 and part 2. (Including Single, Global, First-Fit, Best-Fit, Worst-Fit) 5%**

    首先使用多核的執行時間比起單核快上不少，而在 GLOAB 時我發現我的電腦在全部的矩陣大小都是一萬時 print 不出 core 切換的資訊，可能是電腦的核心比較好的原因。

    而 First-Fit、Best-Fit 和 Worst-Fit 三個排法中，在輸入為 10 的時候，最快的是 Worst-Fitst，不過這是因為他有一個 TASK 是給額外的 core 做，因此比較快也是蠻合理的。

- **Analyze and compare the characteristic of the three different partition methods (First-Fit, Best-Fit, Worst-Fit) 5%**

  在 10 個 thread 中，First-Fit 是唯一能夠全部排進去的排法，最滿的利用率是 0.9957，最少的則是 0.8317，接下來 Best-Fit 的排法沒辦法將最後一個任務排程，最大的利用率是 0.8804，最少的則是 0.8345，最後的 Worst-Fit 則是無法將倒數第二個任務排程，最大的利用率是 0.9152，最少的則是 0.6499，First-Fit 的排法跟進來的任務順序有很大的關係，有可能是剛好塞的下，Best-Fit 的排法是先將最大使用率的 core 塞滿，理論上感覺可以省下最多的利用率，但從結果來看分配還蠻平均的，我認為主要是任務的利用率都不算小，因此常發生塞不下的情況，最後，Worst-Fit 的排法是先排進利用率最小的 core，理論上分配能夠最平均，但結果看來差挺多的，主要是從第七個任務(Thread 6)開始放的任務量差蠻多的，只看前六個任務的話還蠻平均的。

- **Analyze and compare the response time of the program, with two different schedulers. (FIFO with FF, RR with FF) 5%**

  FIFO 的 context switch 次數明顯比 RR 少非常多，這是因為 FIFO 在執行完當前任務前不會切換給下一個任務，而 RR 則是每個任務會輪流執行，比較執行時間的話，我的結果是 RR 比 FIFO 快 0.5 秒，不過這個差距有可能是受到電腦當時的背景程序影響，否則我認為 context switch 比較少的 FIFO 應該會快一點才對。