Crediting :

The number of created threads must be twice as the number of CPUs, and the number of CPUs should be more than 3.

[Global Scheduling. 30%]

Describe how to implement multithread by using OpenMP **10%**

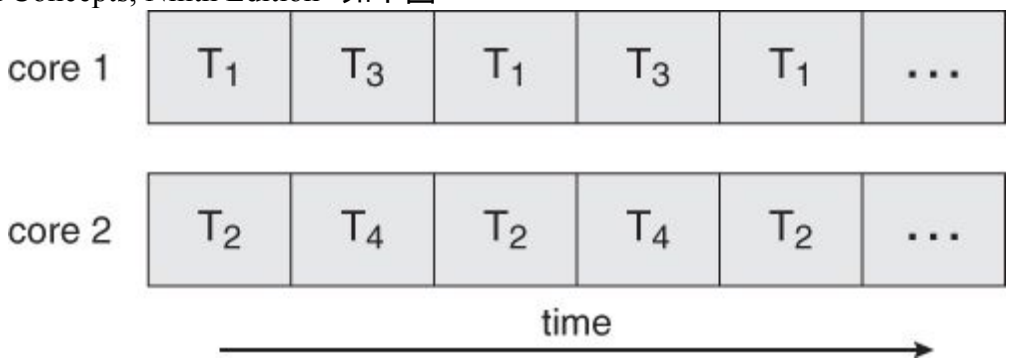
我以SourceCode程式碼為例子，底下顯示我修改程式的部份，實際的程式碼依然會附上
https://www.diffchecker.com/oj5fkdhv
我先說明我把變數

```
float a[100000], b[100000], sum;
```

改成

```
long a[100000], b[100000],sum;
```

由於float的精準度不夠導致sum += (a[i] * b[i]);再做時某些低位元的值會不准
再者參考至聖經 Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition "如下圖

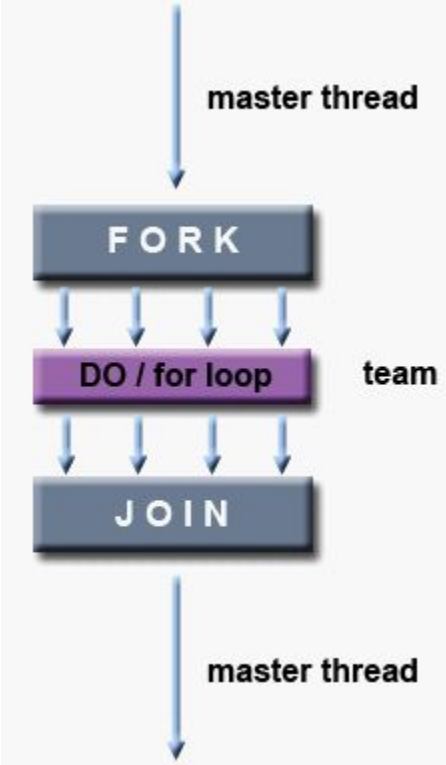


實際上thread被排進core 1 與core 2每次執行排的情形不一定都會一樣，也不一定會按照順序，所以我在執行時會發現執行的結果會變動。

```
        sum = 0;
         #pragma omp parallel private(i) reduction(+:sum)
         {
              cpu_alloc[omp_get_thread_num()]= sched_getcpu();
              #pragma omp for
              for (i=0; i < n; i++)
              {
                  sum +=  (a[i] * b[i]);
                  if(sched_getcpu()!=cpu_alloc[omp_get_thread_num()])
                  {
                          printf("The thread %d is moved from CPU%d to CPU%d \n",
omp_get_thread_num(), cpu_alloc[omp_get_thread_num()], sched_getcpu());
                          cpu_alloc[omp_get_thread_num()]= sched_getcpu();
                          migration_counter++;
                  }
              }
```

```
}
```

為了讓它平行處理所以加上`#pragma omp parallel` 而後面的 `private(i)` 目的是用讓 i 變數複製到 每一個thread，最重要的是`reduction(+:sum)` 要讓sum再每一個thread之中的運算結果在家總起來，否則sum的結果會出錯，因為如果sum用成shared(sum)，雖然每個thread會共同使用sum變數，但有可能會陰同時存取sum的值造成出錯。
最後再加上`#pragma omp for` 目的是為了那每個thread 分配去執行for迴圈，如果沒加此行就會造成for迴圈重複執行



Describe how to estimate task migration **10%**

透過以下程式敘述

```
cpu_alloc[omp_get_thread_num()]= sched_getcpu();
if(sched_getcpu()!=cpu_alloc[omp_get_thread_num()])
```

`omp_get_thread_num()`可以得知目前該thread的thread id
`sched_getcpu()`可以得知目前thread是由那一個cpu core執行
利用cpu_alloc陣列來存放thread使用那一顆cpu core執行
假設判斷到目前的cpu core number不等於上一個cpu core number
表示該thread由某個cpu core migration至另一個cpu core

Show the result of task migration and describe why task is migrated **10%**

我用簡單的例子來說明 Global Scheduling 可能產生的 migration 原因兩點如下：

1. CPU way 被其他 thread 佔住資源為了load balance導致migration
2.

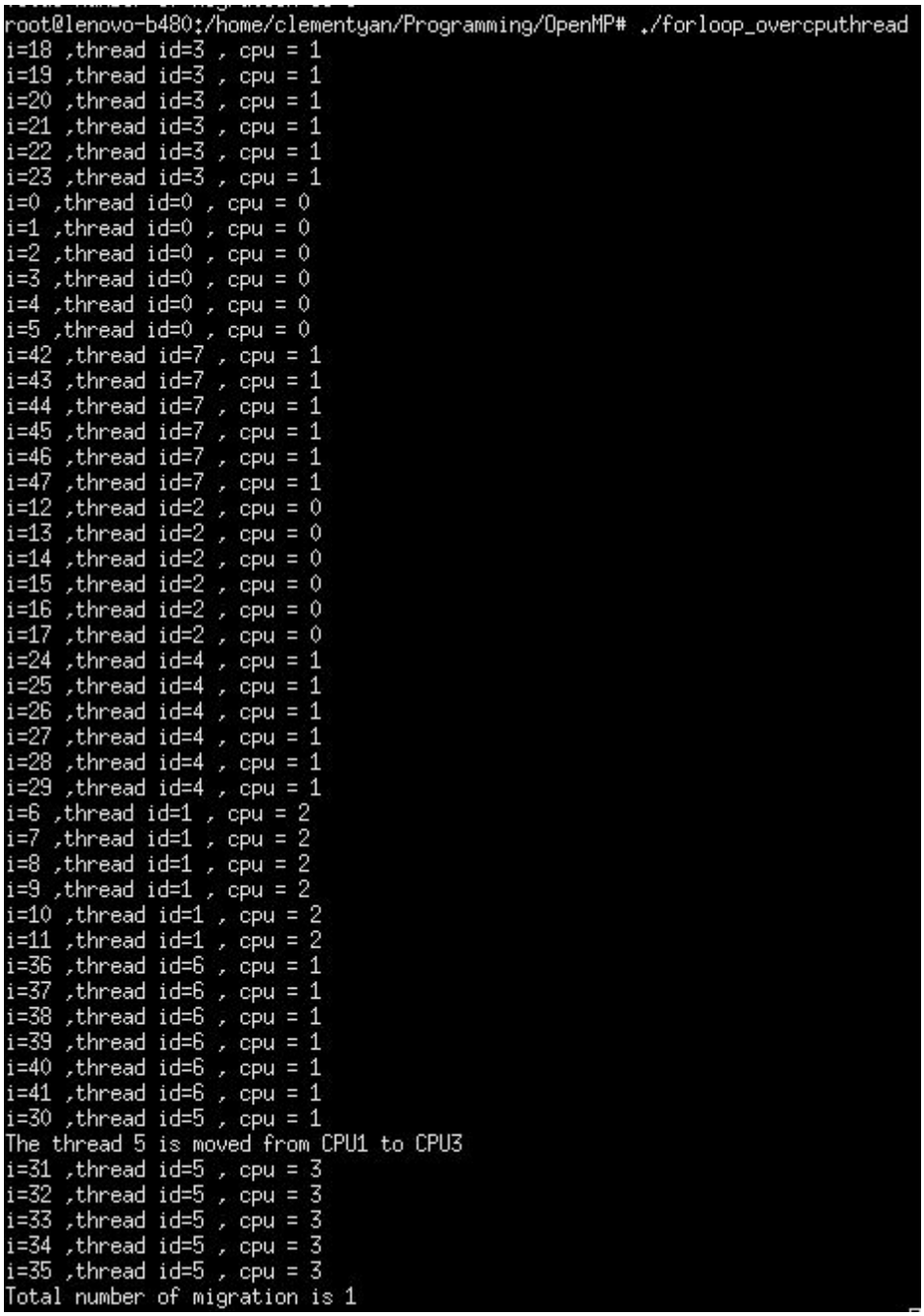
程式碼如下(使用gcc version 4.9.2, 4 cpu way, process fork 8 threads)

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>

int sched_getcpu(void);

int main (int argc, char *argv[]) {
        int set_num_threads = 8;
        int cpu_alloc[set_num_threads];
        int migration_counter=0;
        omp_set_num_threads(set_num_threads);
        cpu_alloc[omp_get_thread_num()]= sched_getcpu();
        #pragma omp parallel
        {
                cpu_alloc[omp_get_thread_num()]= sched_getcpu();
                int i;
                #pragma omp for
                for(i=0;i<48;i++){
//                      printf("i=%d thread id=%d\n",i,omp_get_thread_num());
                        printf("i=%d ,thread id=%d , cpu =
%d\n",i,omp_get_thread_num(),sched_getcpu());
                        if(sched_getcpu()!=cpu_alloc[omp_get_thread_num()])
                        {
                                printf("The thread %d is moved from CPU%d to CPU%d \n",
omp_get_thread_num(), cpu_alloc[omp_get_thread_num()], sched_getcpu());
                                cpu_alloc[omp_get_thread_num()]= sched_getcpu();
                                migration_counter++;
                        }
                }
        }
        printf("Total number of migration is %d \n", migration_counter);
        return 0;
}
```

輸出結果如下



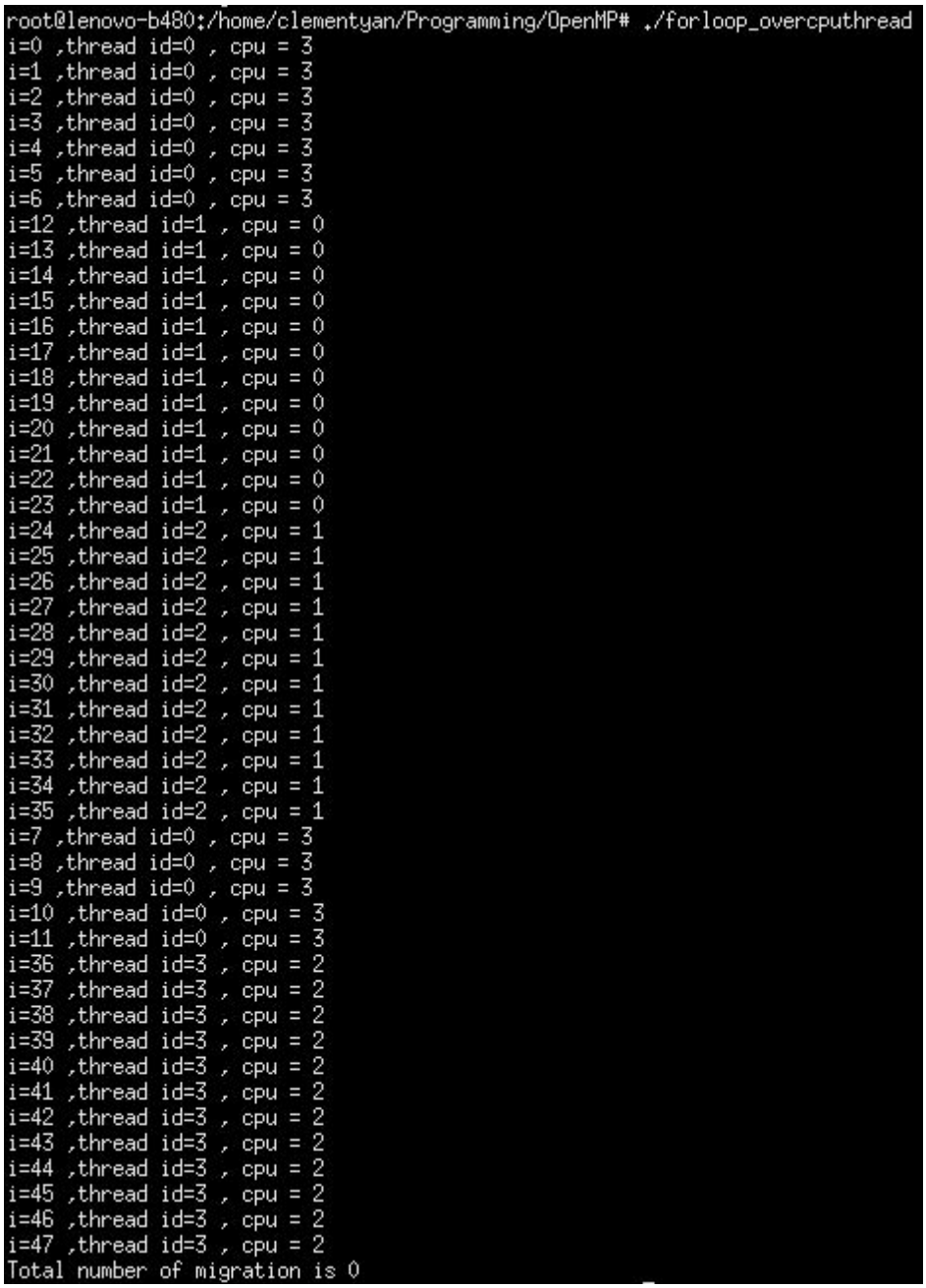
圖一

從輸出結果可以看到 i=36~41 thread id=6在cpu 1(cpu 1 way)上執行，而在i=30 ithread id=5是在cpu 1執行，但在 i=31~35 thread id=5卻migration至CPU 3，我推測可能是為了要讓cpu load balance上面我們可以看到cpu 3都還為被使用

解決方式

將 `set_num_threads = 8;` 改成與你cpu way 的數量一樣 =4

輸出如下



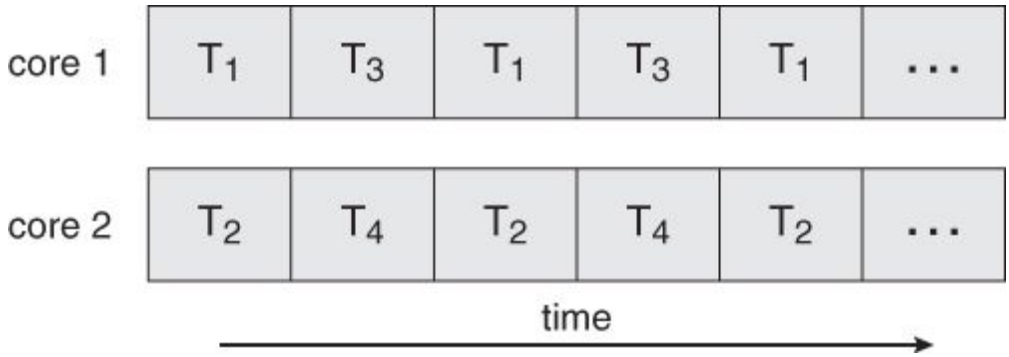
圖二

不論我執行幾次它都不會出現任何migration。

底下我做更詳細的解釋：

參考至聖經 Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 1 p.16

假設透過 OpenMP API fork出來的 thread有四個thread(`set_num_threads = 4;`) 而實際你的 cpu 只有兩個core， 排進core的方式示意圖如下



圖三

但實際上其實thread再排並不會按照順序

回過頭來看圖一，依照我的cpu他是如下排列

core 0：T_3 T_7 T_4 T_6 T_5

core 1：T_0 T_2

core 2：T_1

core 3：T_5

(實際上我的cpu 是 2 core, 2 SMT per core，我暫且用core來代替名稱)

原先T_5是被排進core 0但發現core 3 使用率低為了cpu load balance 故把T_5 migration至core 3

而圖二的情形，依照我的cpu他是如下排列

core 0：T_1

core 1：T_2

core 2：T_3

core 3：T_0

因此每個cpu core 使用率較平衡故不會有migration

3. 另外一個造成migration次數較多的原因，因為cpu 在scheduling thread 到cpu way，仍然會有load balance原因造成migration，而如果在巢狀for迴圈當中平行化處理只平行內層for迴圈將會大大增加 migration的情形，如下

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>

int sched_getcpu(void);

int main (int argc, char *argv[]) {
        int set_num_threads = 5;
        int cpu_alloc[set_num_threads];
        int migration_counter=0;
        omp_set_num_threads(set_num_threads);
        cpu_alloc[omp_get_thread_num()]= sched_getcpu();
int j;
      #pragma omp parallel
      {
              cpu_alloc[omp_get_thread_num()]= sched_getcpu();
              #pragma omp for
        for(j=0;j<10;j++)
```

```
        {


                int i;

                for(i=0;i<10;i++){
                        printf("j=%d, i=%d ,thread id=%d , cpu =
%d\n",j,i,omp_get_thread_num(),sched_getcpu());
                        if(sched_getcpu()!=cpu_alloc[omp_get_thread_num()])
                        {
                                printf("The thread %d is moved from CPU%d to CPU%d \n",
omp_get_thread_num(), cpu_alloc[omp_get_thread_num()], sched_getcpu());
                                cpu_alloc[omp_get_thread_num()]= sched_getcpu();
                                migration_counter++;
                        }
                }
        }
        }

        printf("Total number of migration is %d \n", migration_counter);
        return 0;
}
```

```
j=4, i=0 ,thread id=2 , cpu = 2
j=4, i=1 ,thread id=2 , cpu = 2
j=6, i=0 ,thread id=3 , cpu = 3
j=6, i=1 ,thread id=3 , cpu = 3
j=6, i=2 ,thread id=3 , cpu = 3
j=8, i=0 ,thread id=4 , cpu = 0
j=8, i=1 ,thread id=4 , cpu = 0
j=8, i=2 ,thread id=4 , cpu = 0
j=8, i=3 ,thread id=4 , cpu = 0
j=8, i=4 ,thread id=4 , cpu = 0
j=8, i=5 ,thread id=4 , cpu = 0
j=8, i=6 ,thread id=4 , cpu = 0
j=8, i=7 ,thread id=4 , cpu = 0
j=8, i=8 ,thread id=4 , cpu = 0
j=4, i=2 ,thread id=2 , cpu = 2
j=4, i=3 ,thread id=2 , cpu = 2
j=4, i=4 ,thread id=2 , cpu = 2
j=6, i=3 ,thread id=3 , cpu = 3
j=6, i=4 ,thread id=3 , cpu = 3
j=6, i=5 ,thread id=3 , cpu = 3
j=6, i=6 ,thread id=3 , cpu = 3
j=6, i=7 ,thread id=3 , cpu = 3
j=6, i=8 ,thread id=3 , cpu = 3
j=6, i=9 ,thread id=3 , cpu = 3
j=7, i=0 ,thread id=3 , cpu = 3
j=7, i=1 ,thread id=3 , cpu = 3
j=7, i=2 ,thread id=3 , cpu = 3
j=7, i=3 ,thread id=3 , cpu = 3
j=7, i=4 ,thread id=3 , cpu = 3
j=7, i=5 ,thread id=3 , cpu = 3
j=7, i=6 ,thread id=3 , cpu = 3
j=7, i=7 ,thread id=3 , cpu = 3
```

```
j=1, i=5 ,thread id=0 , cpu = 1
j=1, i=6 ,thread id=0 , cpu = 1
j=1, i=7 ,thread id=0 , cpu = 1
j=1, i=8 ,thread id=0 , cpu = 1
j=1, i=9 ,thread id=0 , cpu = 1
Total number of migration is 3
```

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>

int sched_getcpu(void);

int main (int argc, char *argv[]) {
        int set_num_threads = 5;
        int cpu_alloc[set_num_threads];
        int migration_counter=0;
        omp_set_num_threads(set_num_threads);
        cpu_alloc[omp_get_thread_num()]= sched_getcpu();
        int j;
        for(j=0;j<10;j++)
        {

        #pragma omp parallel
        {
                cpu_alloc[omp_get_thread_num()]= sched_getcpu();
                int i;
                #pragma omp for
                for(i=0;i<10;i++){
                        printf("j=%d, i=%d ,thread id=%d , cpu =
%d\n",j,i,omp_get_thread_num(),sched_getcpu());
                        if(sched_getcpu()!=cpu_alloc[omp_get_thread_num()])
                        {
                                printf("The thread %d is moved from CPU%d to CPU%d \n",
omp_get_thread_num(), cpu_alloc[omp_get_thread_num()], sched_getcpu());
                                cpu_alloc[omp_get_thread_num()]= sched_getcpu();
                                migration_counter++;
                        }
                }
        }
        }

        printf("Total number of migration is %d \n", migration_counter);
        return 0;
}
```

```
j=7, i=6 ,thread id=3 , cpu = 0
The thread 3 is moved from CPU0 to CPU1
j=7, i=7 ,thread id=3 , cpu = 1
j=7, i=8 ,thread id=4 , cpu = 2
j=7, i=9 ,thread id=4 , cpu = 2
j=8, i=0 ,thread id=0   cpu = 3
j=8, i=1 ,thread id=0   cpu = 3
j=8, i=8 ,thread id=4   cpu = 0
j=8, i=9 ,thread id=4   cpu = 0
j=8, i=6 ,thread id=3   cpu = 1
j=8, i=7 ,thread id=3   cpu = 1
j=8, i=2 ,thread id=1   cpu = 2
j=8, i=3 ,thread id=1   cpu = 2
j=8, i=4 ,thread id=2   cpu = 1
The thread 2 is moved from CPU1 to CPU3
j=8, i=5 ,thread id=2   cpu = 3
```

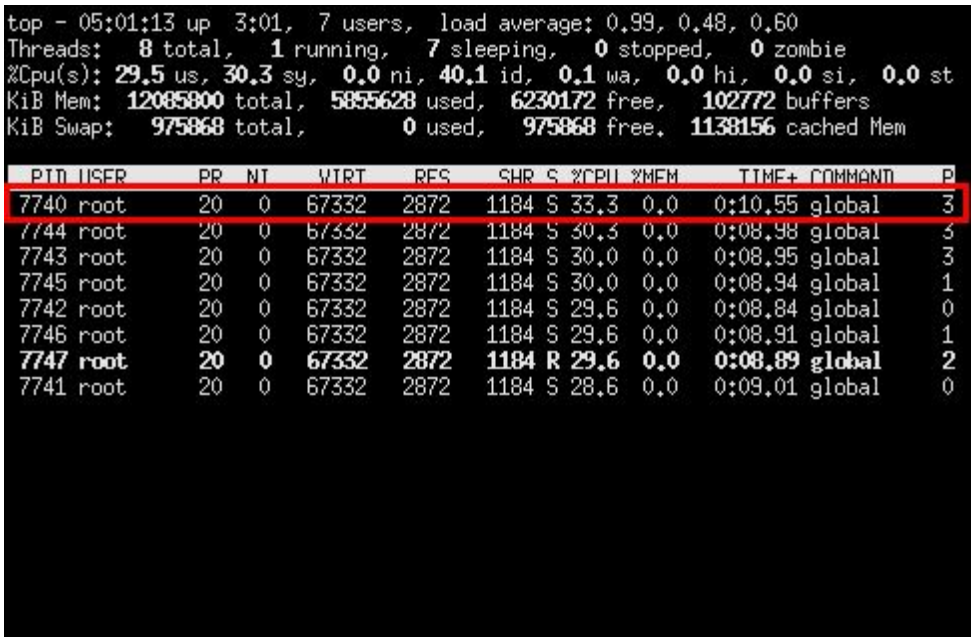
```
The thread 3 is moved from CPU1 to CPU3
j=9, i=7 ,thread id=3 , cpu = 3
j=9, i=4 ,thread id=2 , cpu = 3
The thread 2 is moved from CPU3 to CPU1
Total number of migration is 19
```

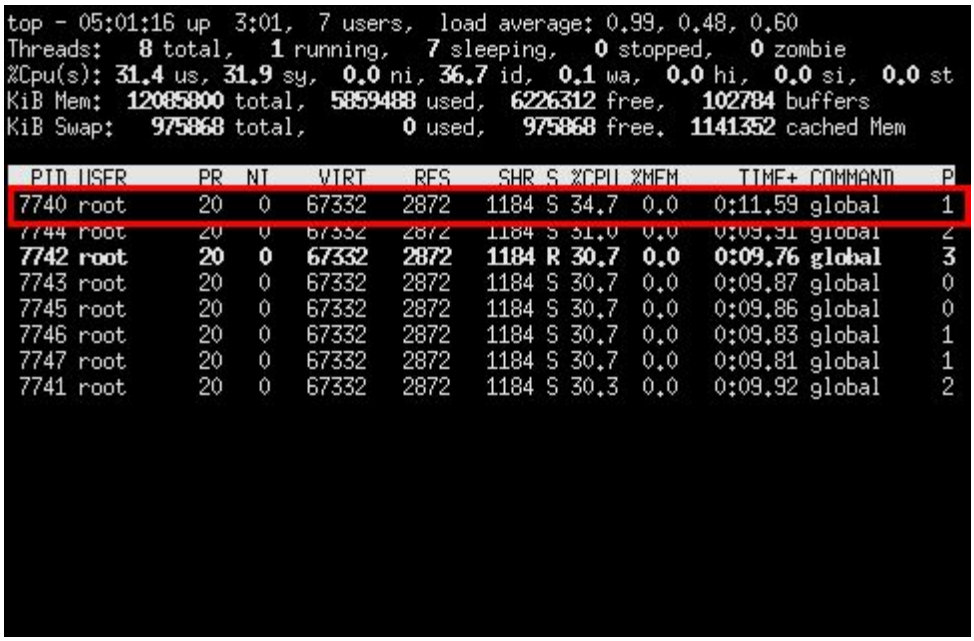
再回到SourceCode.c，從SourceCode.c的程式我發現若程式發生migration次數太多將會造成程式執行過程時間嚴重加長，而為了實驗我刻意將平行化處理寫在內層for迴圈增加migration來做比較，但實際上正確應該把平行處理寫在外層迴圈執行速度才會快

Case 1：process fork出來的thread數量大於cpu 開啟後的 way數量

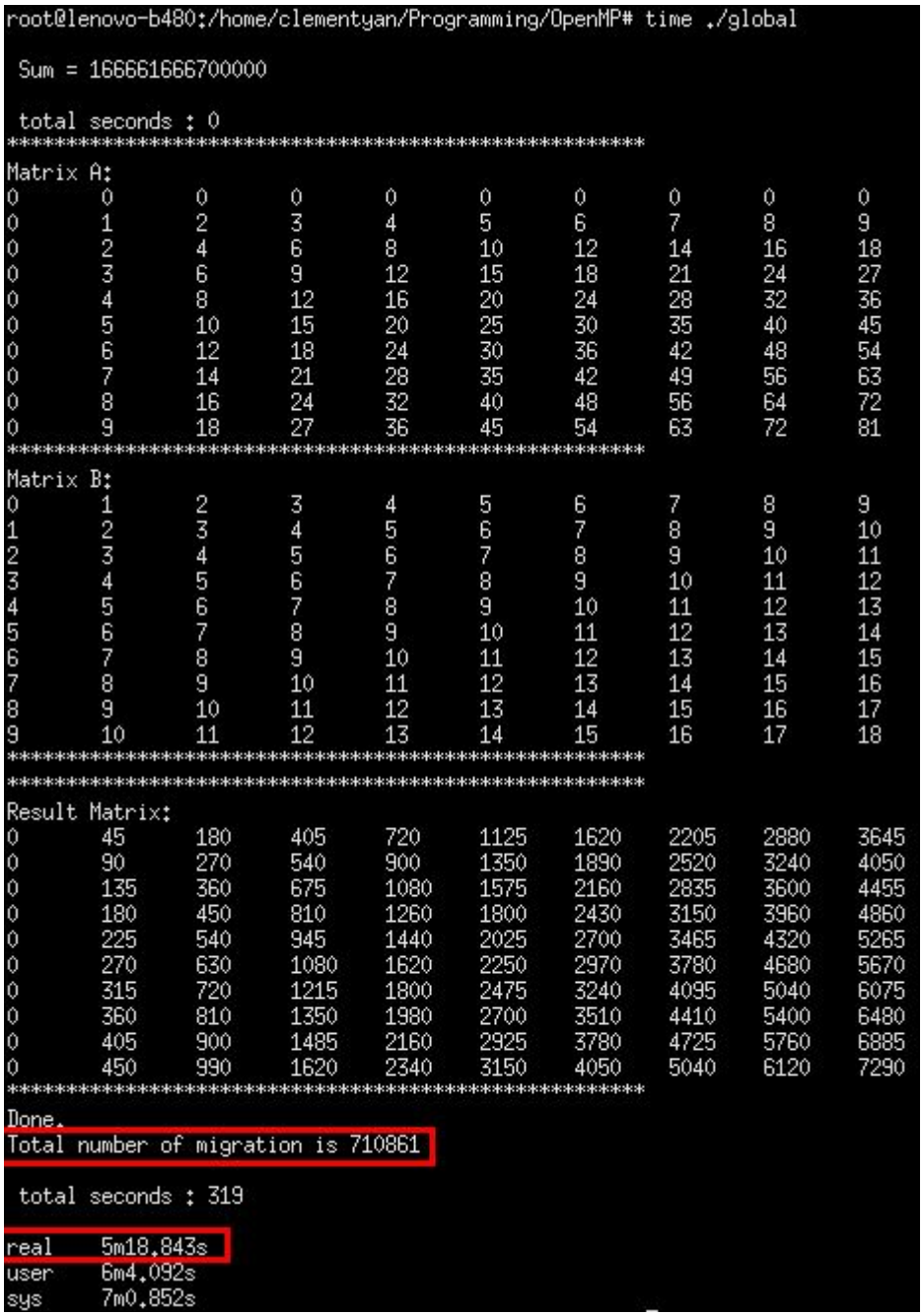
```
top - 05:01:19 up  3:01,  7 users,  load average: 1.39, 0.57, 0.63
Tasks: 241 total,   2 running, 239 sleeping,   0 stopped,   0 zombie
%Cpu(s): 30.5 us, 31.8 sy,  0.0 ni, 37.6 id,  0.1 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  12085800 total,  5860568 used,  6225232 free,   102796 buffers
KiB Swap:   975868 total,        0 used,   975868 free.  1140268 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 7740 root      20   0   67332   2872   1184 R 251.0  0.0   1:27.33 global
 5548 clement+  20   0  784836 144636  44216 S   4.0  1.2   0:18.80 shutter
 1263 root      20   0  171696  29684  14028 S   1.7  0.2   2:47.75 Xorg
 1419 clement+  20   0 3370996 335296 121108 S   1.7  2.8   7:38.16 chromium
 1506 clement+  20   0 1366288  99120  61652 S   1.3  0.8   1:08.34 chromium
 1874 clement+  20   0 1471060 197832  69680 S   1.3  1.6   1:20.59 chromium
 3690 clement+  20   0 1496104 208248  90080 S   0.7  1.7   1:44.17 chromium
    7 root      20   0       0      0      0 S   0.3  0.0   0:09.92 rcu_preempt
 1322 clement+  20   0  367484   9448   6236 S   0.3  0.1   0:44.76 ibus-daemon
 1341 clement+  20   0  116572  20180  15472 S   0.3  0.2   0:18.03 icewm
 1346 clement+  20   0  404060  29040  23520 S   0.3  0.2   0:17.25 ibus-ui-gt+
 1519 clement+  20   0 1366328 119812  47452 S   0.3  1.0   0:30.75 chromium
 1773 clement+  20   0 1425460 163584  63064 S   0.3  1.4   1:01.04 chromium
 1925 clement+  20   0 1448440 170248  63816 S   0.3  1.4   1:04.31 chromium
 1927 clement+  20   0 2388644 210436  70504 S   0.3  1.7   0:54.90 chromium
 3054 clement+  20   0 1484196 191584  75068 S   0.3  1.6   1:47.51 chromium
 3994 clement+  20   0 1484992 195380  84216 S   0.3  1.6   1:00.08 chromium
```





途中可以看到 thread PID 7740 由 cpu3 migration至cpu 1

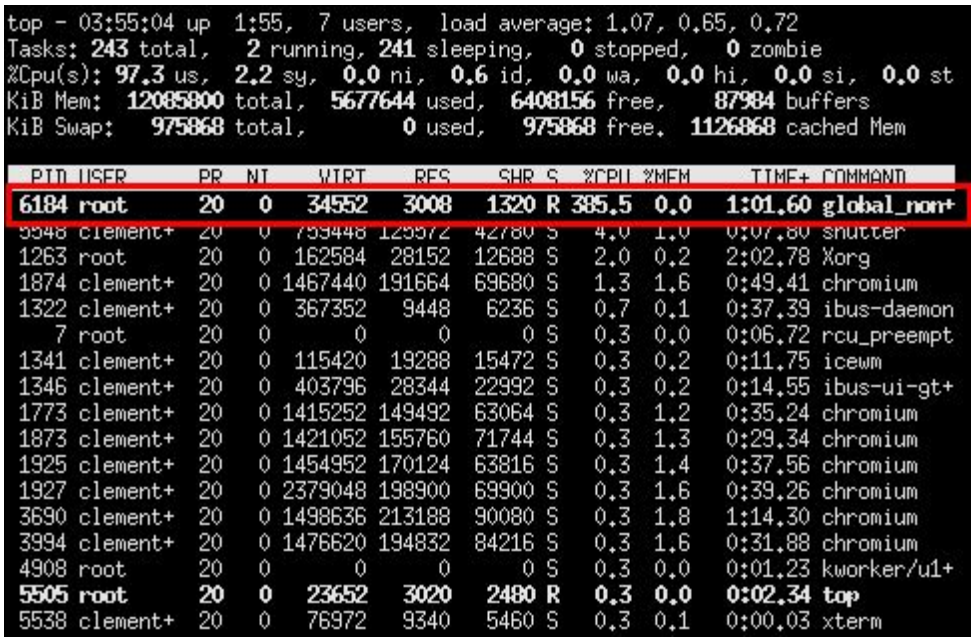


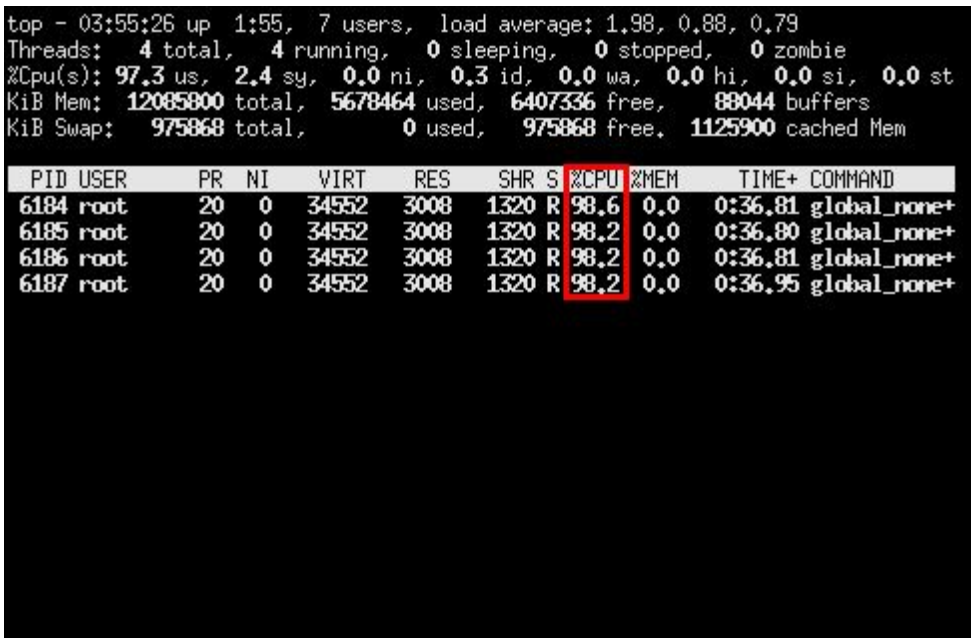
我的程式碼 process fork出來的thread共五個，cpu開啟的way四個，結果我們可以發現它migration的次數發長的高，造成執行respontime延長，再次強調我刻意把平行處理寫在內層迴圈讓它migration次數增加，以下程式皆是如此

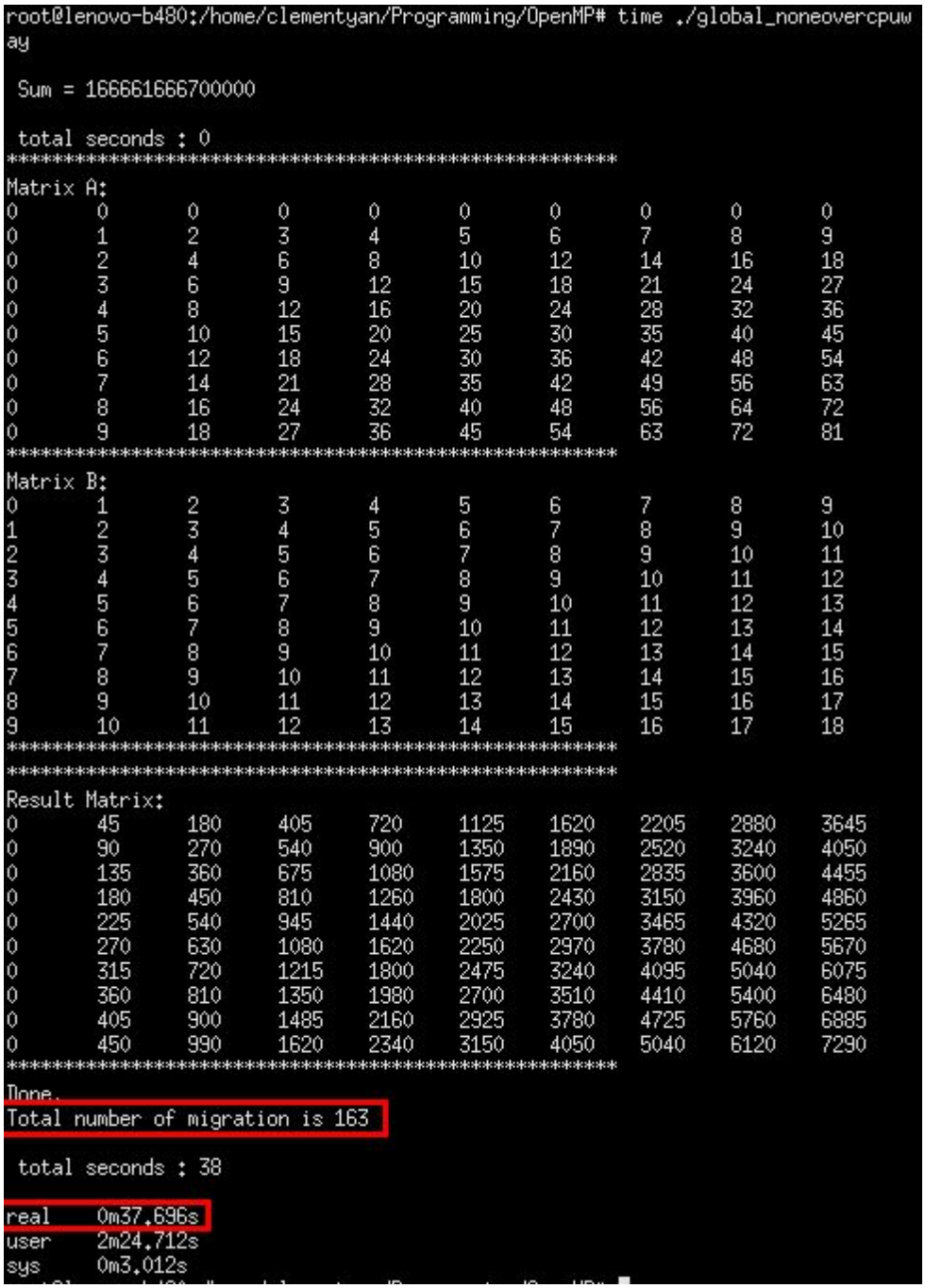
程式碼連結如下網址

https://gist.github.com/clementyan/8dafee88d7cc7117731cf68f2130cfbc

Case 2：process fork出來的thread數量小於cpu 開啟後的 way數量


top - 03:55:04 up 1:55, 7 users, load average: 1.07, 0.65, 0.72
Tasks: 243 total, 2 running, 241 sleeping, 0 stopped, 0 zombie
%Cpu(s): 97.3 us, 2.2 sy, 0.0 ni, 0.6 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 12085800 total, 5677644 used, 6408156 free, 87984 buffers
KiB Swap: 975868 total, 0 used, 975868 free. 1126868 cached Mem
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
6184 root 20 0 34552 3008 1320 R 385.5 0.0 1:01.60 global_non+
5548 clement+ 20 0 759448 125572 42780 S 4.0 1.0 0:07.80 shutter
1263 root 20 0 162584 28152 12688 S 2.0 0.2 2:02.78 Xorg
1874 clement+ 20 0 1467440 191664 69680 S 1.3 1.6 0:49.41 chromium
1322 clement+ 20 0 367352 9448 6236 S 0.7 0.1 0:37.39 ibus-daemon
7 root 20 0 0 0 0 S 0.3 0.0 0:06.72 rcu_preempt
1341 clement+ 20 0 115420 19288 15472 S 0.3 0.2 0:11.75 icewm
1346 clement+ 20 0 403796 28344 22992 S 0.3 0.2 0:14.55 ibus-ui-gt+
1773 clement+ 20 0 1415252 149492 63064 S 0.3 1.2 0:35.24 chromium
1873 clement+ 20 0 1421052 155760 71744 S 0.3 1.3 0:29.34 chromium
1925 clement+ 20 0 1454952 170124 63816 S 0.3 1.4 0:37.56 chromium
1927 clement+ 20 0 2379048 198900 69900 S 0.3 1.6 0:39.26 chromium
3690 clement+ 20 0 1498636 213188 90080 S 0.3 1.8 1:14.30 chromium
3994 clement+ 20 0 1476620 194832 84216 S 0.3 1.6 0:31.88 chromium
4908 root 20 0 0 0 0 S 0.3 0.0 0:01.23 kworker/u1+
5505 root 20 0 23652 3020 2480 R 0.3 0.0 0:02.34 top
5538 clement+ 20 0 76972 9340 5460 S 0.3 0.1 0:00.03 xterm


top - 03:55:26 up 1:55, 7 users, load average: 1.98, 0.88, 0.79
Threads: 4 total, 4 running, 0 sleeping, 0 stopped, 0 zombie
%Cpu(s): 97.3 us, 2.4 sy, 0.0 ni, 0.3 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 12085800 total, 5678464 used, 6407336 free, 88044 buffers
KiB Swap: 975868 total, 0 used, 975868 free. 1125900 cached Mem
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
6184 root 20 0 34552 3008 1320 R 98.6 0.0 0:36.81 global_none+
6185 root 20 0 34552 3008 1320 R 98.2 0.0 0:36.80 global_none+
6186 root 20 0 34552 3008 1320 R 98.2 0.0 0:36.81 global_none+
6187 root 20 0 34552 3008 1320 R 98.2 0.0 0:36.95 global_none+



結果來看cpu 使用率提昇，執行所需的時間也下降須多，migration次數只有163次

程式碼如下

https://gist.github.com/clementyan/0a4a10bb32a906c4e91cb7b80d908e23

[Partition Scheduling. 30%]

Describe how to implement partition scheduling 20%

我將程式馬加上下列敘述，使thread綁定cpu core如下

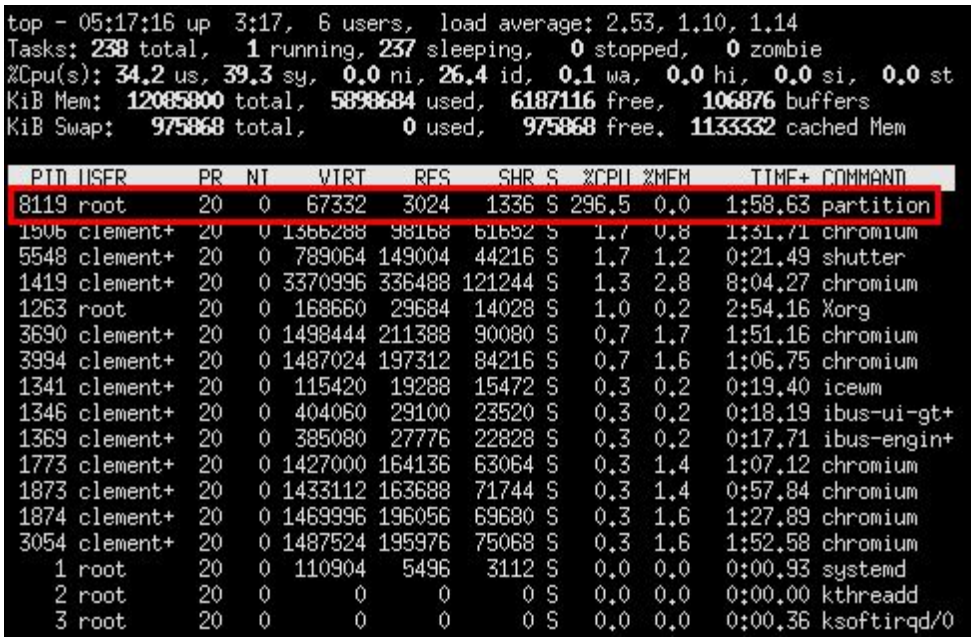
core 0：T_0 T_1

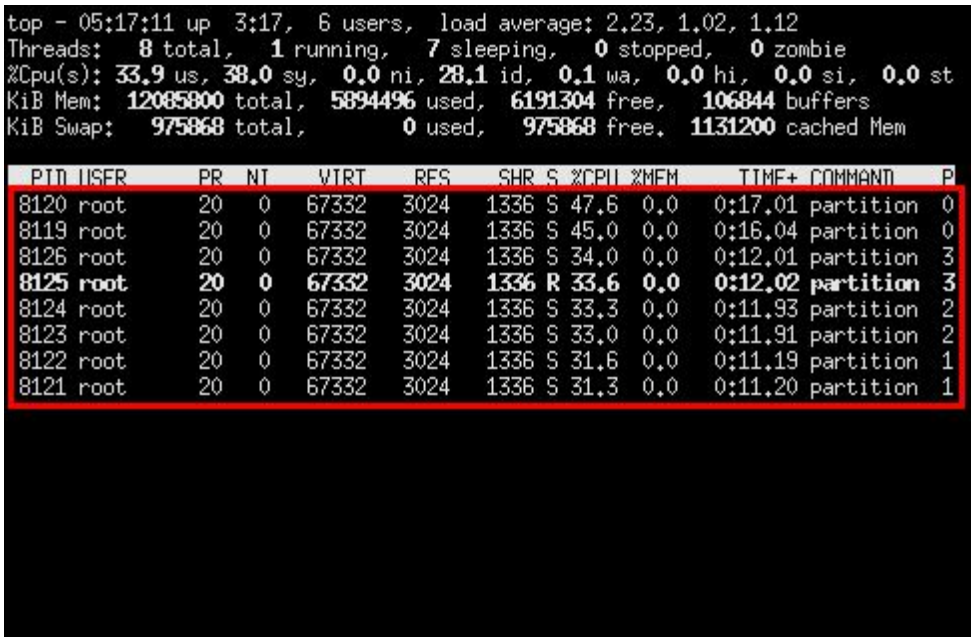
core 1：T_2 T_3

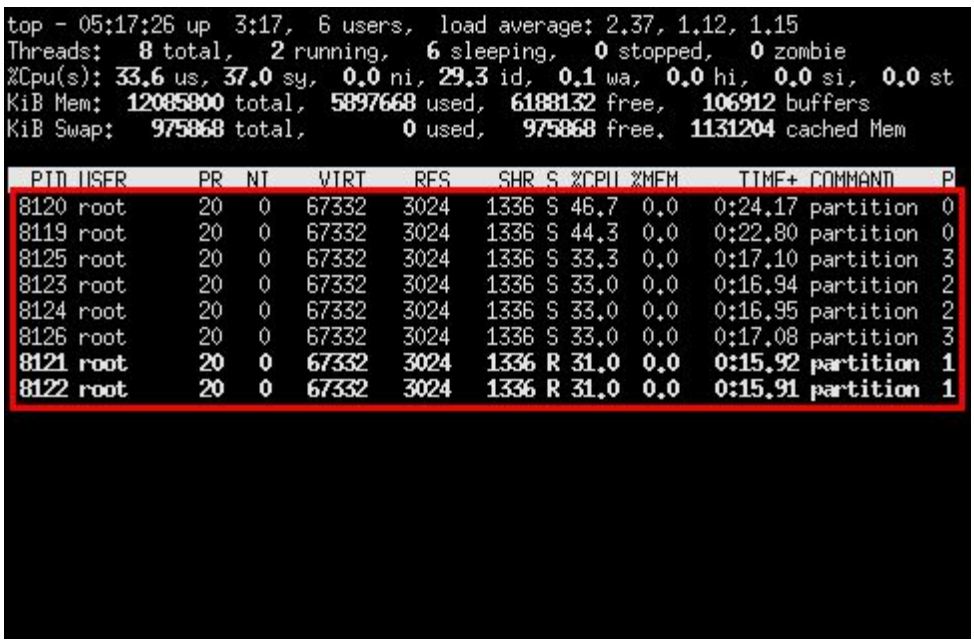
core 2：T_4 T_5

core 3：T_6 T_7

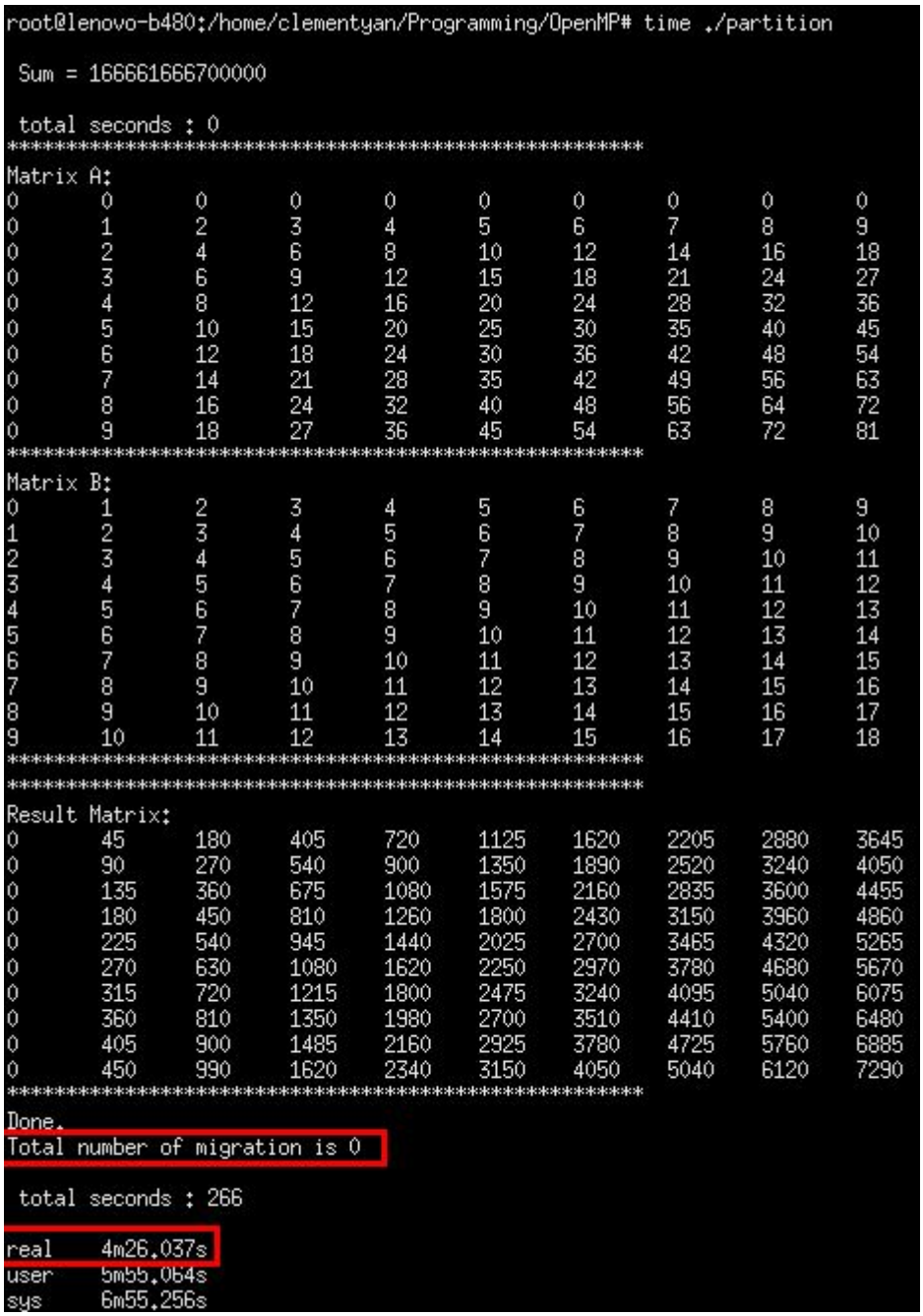
```
#pragma omp parallel for num_threads(set_num_threads) private(PID,ret,set)
for(i=0;i<omp_get_num_threads();i++)
{
        PID=getpid()+omp_get_thread_num();
        CPU_ZERO(&set);
        CPU_SET(i/2,&set);
        ret=sched_setaffinity(PID,sizeof(cpu_set_t),&set);
}
```







從圖中可以看到thread被我綁到cpu way不會有migration現象



執行結果沒有migration因為我綁住每個thread至固定的cpu way，執行所需的時間也相較於global會migration的情形低

Show the scheduling states of tasks
10%

[Scheduler Implementation. 20%]

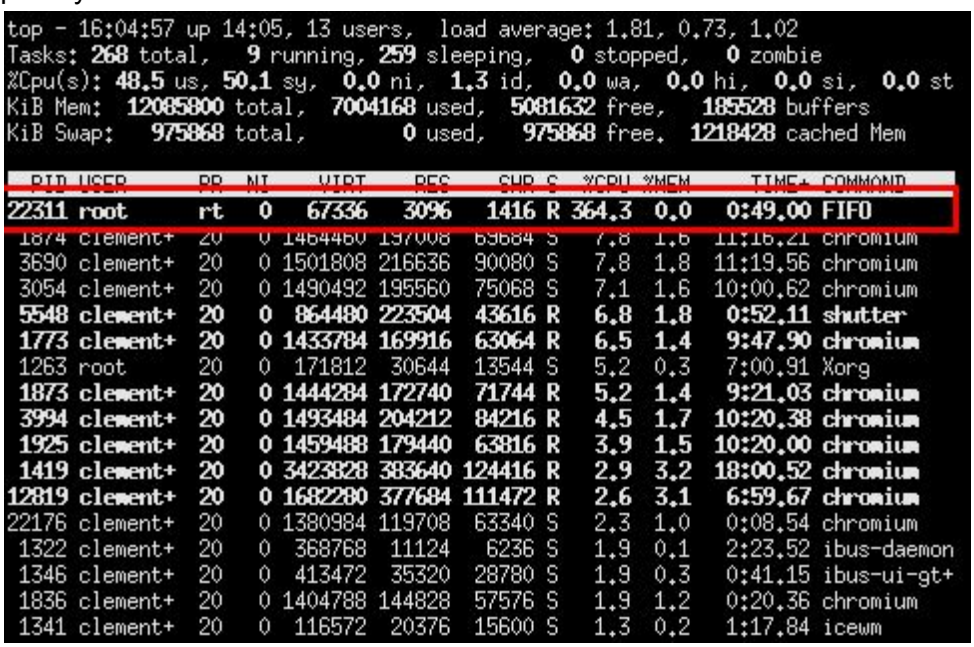
Describe how to implement the scheduler setting (FIFO, RR, Default) 10%

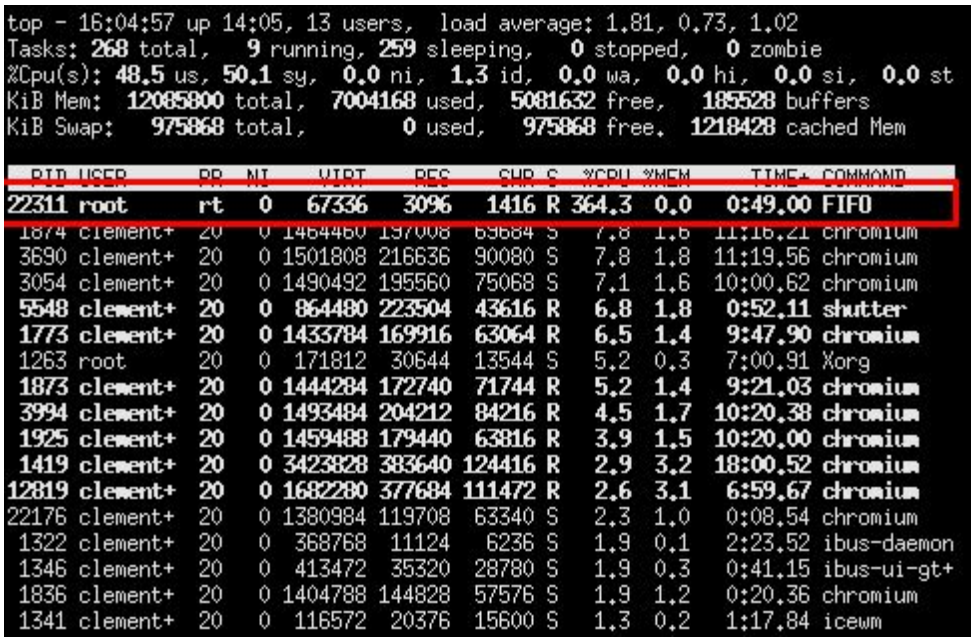
利用以下程式來設定scheduling

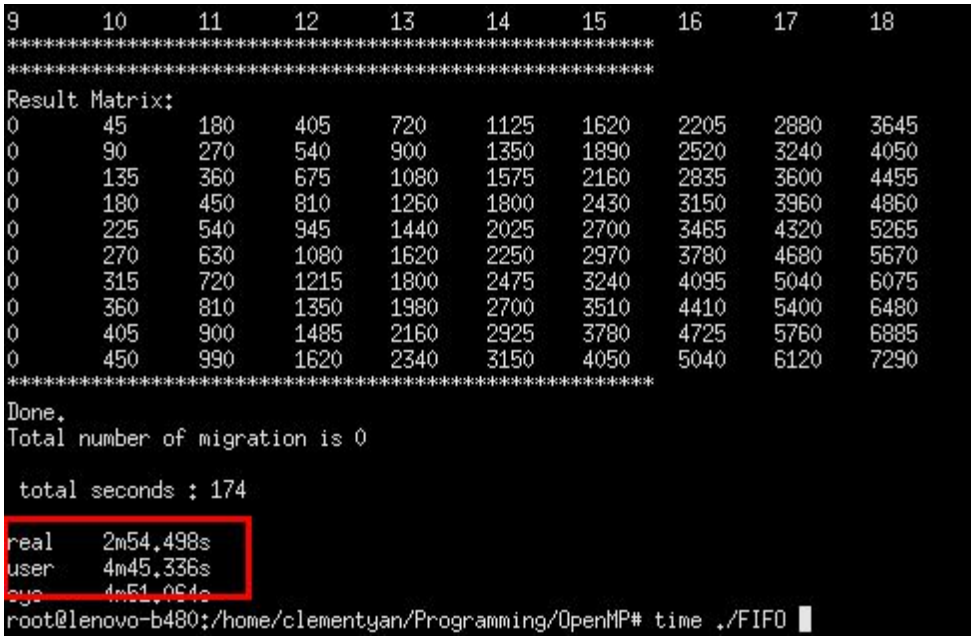
FIFO：

```
struct sched_param sp;
sp.sched_priority=sched_get_priority_max(SCHED_FIFO);
ret=sched_setscheduler(0,SCHED_FIFO,&sp);
```

執行結果整個電腦速度會嚴重下降，應該是因為我該程式改用FIFO real-time，並提高他的priority



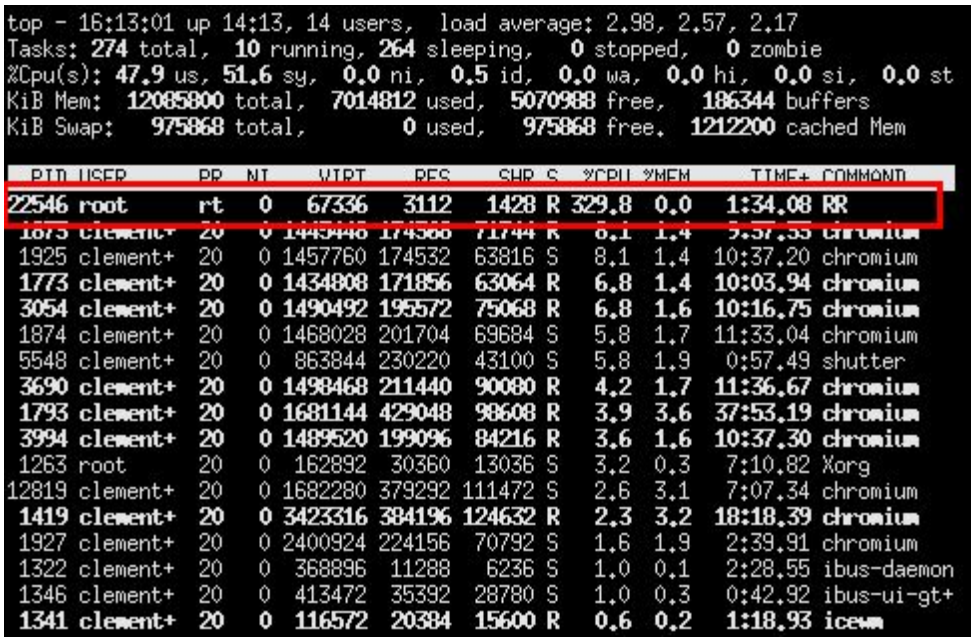


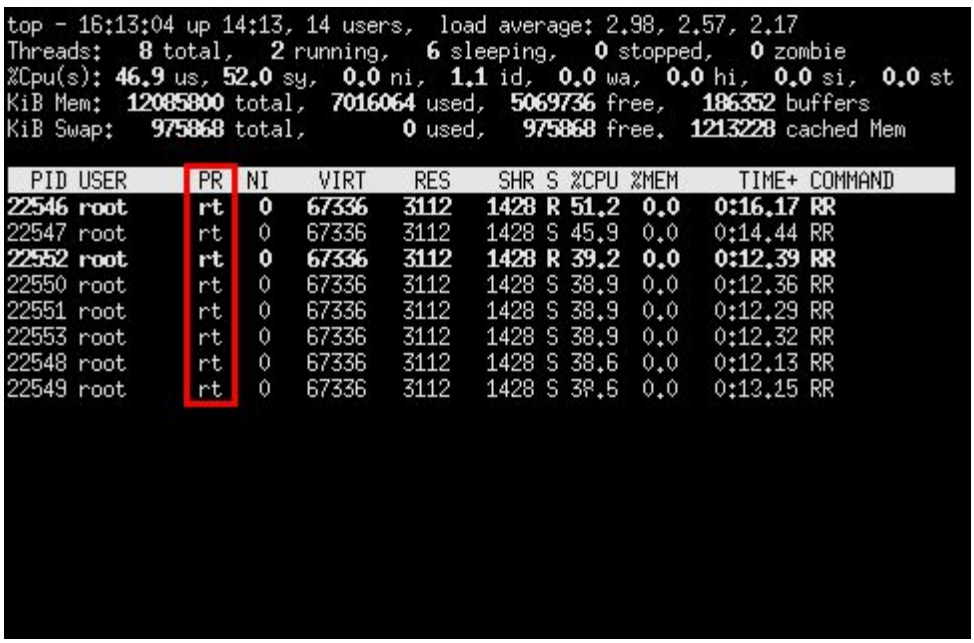


FIFO.c這隻程式我是沿用partition.c，可原先沒使用FIFO scheduling，使用FIFO scheduling加快執行所需要的時間

RR：

```
struct sched_param sp;
sp.sched_priority=sched_get_priority_max(SCHED_RR);
ret=sched_setscheduler(0,SCHED_RR,&sp);
```


top - 16:13:01 up 14:13, 14 users, load average: 2.98, 2.57, 2.17
Tasks: 274 total, 10 running, 264 sleeping, 0 stopped, 0 zombie
%Cpu(s): 47.9 us, 51.6 sy, 0.0 ni, 0.5 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 12085800 total, 7014812 used, 5070988 free, 186344 buffers
KiB Swap: 975868 total, 0 used, 975868 free. 1212200 cached Mem
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
22546 root rt 0 67336 3112 1428 R 329.8 0.0 1:34.08 RR
1925 clement+ 20 0 1457760 174532 63816 S 8.1 1.4 10:37.20 chromium
1773 clement+ 20 0 1434808 171856 63064 R 6.8 1.4 10:03.94 chromium
3054 clement+ 20 0 1490492 195572 75068 R 6.8 1.6 10:16.75 chromium
1874 clement+ 20 0 1468028 201704 69684 S 5.8 1.7 11:33.04 chromium
5548 clement+ 20 0 863844 230220 43100 S 5.8 1.9 0:57.49 shutter
3690 clement+ 20 0 1498468 211440 90080 R 4.2 1.7 11:36.67 chromium
1793 clement+ 20 0 1681144 429048 98608 R 3.9 3.6 37:53.19 chromium
3994 clement+ 20 0 1489520 199096 84216 R 3.6 1.6 10:37.30 chromium
1263 root 20 0 162892 30360 13036 S 3.2 0.3 7:10.82 Xorg
12819 clement+ 20 0 1682280 379292 111472 S 2.6 3.1 7:07.34 chromium
1419 clement+ 20 0 3423316 384196 124632 R 2.3 3.2 18:18.39 chromium
1927 clement+ 20 0 2400924 224156 70792 S 1.6 1.9 2:39.91 chromium
1322 clement+ 20 0 368896 11288 6236 S 1.0 0.1 2:28.55 ibus-daemon
1346 clement+ 20 0 413472 35392 28780 S 1.0 0.3 0:42.92 ibus-ui-gt+
1341 clement+ 20 0 116572 20384 15600 R 0.6 0.2 1:18.93 icewm


top - 16:13:04 up 14:13, 14 users, load average: 2.98, 2.57, 2.17
Threads: 8 total, 2 running, 6 sleeping, 0 stopped, 0 zombie
%Cpu(s): 46.9 us, 52.0 sy, 0.0 ni, 1.1 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 12085800 total, 7016064 used, 5069736 free, 186352 buffers
KiB Swap: 975868 total, 0 used, 975868 free. 1213228 cached Mem
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
22546 root rt 0 67336 3112 1428 R 51.2 0.0 0:16.17 RR
22547 root rt 0 67336 3112 1428 S 45.9 0.0 0:14.44 RR
22552 root rt 0 67336 3112 1428 R 39.2 0.0 0:12.39 RR
22550 root rt 0 67336 3112 1428 S 38.9 0.0 0:12.36 RR
22551 root rt 0 67336 3112 1428 S 38.9 0.0 0:12.29 RR
22553 root rt 0 67336 3112 1428 S 38.9 0.0 0:12.32 RR
22548 root rt 0 67336 3112 1428 S 38.6 0.0 0:12.13 RR
22549 root rt 0 67336 3112 1428 S 38.6 0.0 0:13.15 RR

```
9       10      11      12      13      14      15      16      17      18
************************************************************
************************************************************
Result Matrix:
0       45      180     405     720     1125    1620    2205    2880    3645
0       90      270     540     900     1350    1890    2520    3240    4050
0       135     360     675     1080    1575    2160    2835    3600    4455
0       180     450     810     1260    1800    2430    3150    3960    4860
0       225     540     945     1440    2025    2700    3465    4320    5265
0       270     630     1080    1620    2250    2970    3780    4680    5670
0       315     720     1215    1800    2475    3240    4095    5040    6075
0       360     810     1350    1980    2700    3510    4410    5400    6480
0       405     900     1485    2160    2925    3780    4725    5760    6885
0       450     990     1620    2340    3150    4050    5040    6120    7290
************************************************************
Done.
Total number of migration is 0

 total seconds : 174

real    2m53.933s
user    4m47.896s
sys     4m48.212s
root@lenovo-b480:/home/clementyan/Programming/OpenMP#
```

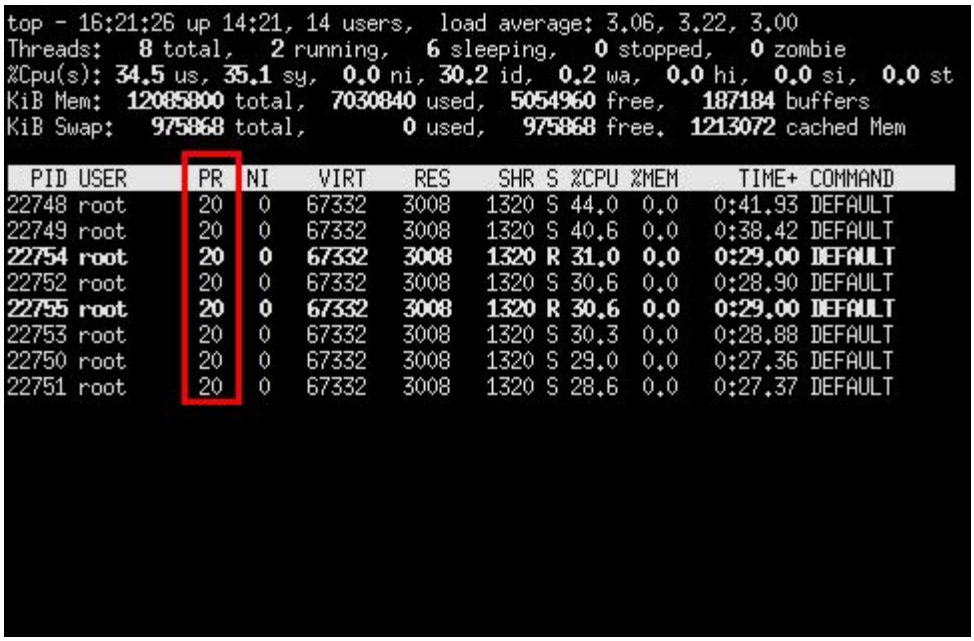
RR.c這隻程式我是沿用partition.c，可原先沒使用RR scheduling，使用RR scheduling加快執行所需要的時間

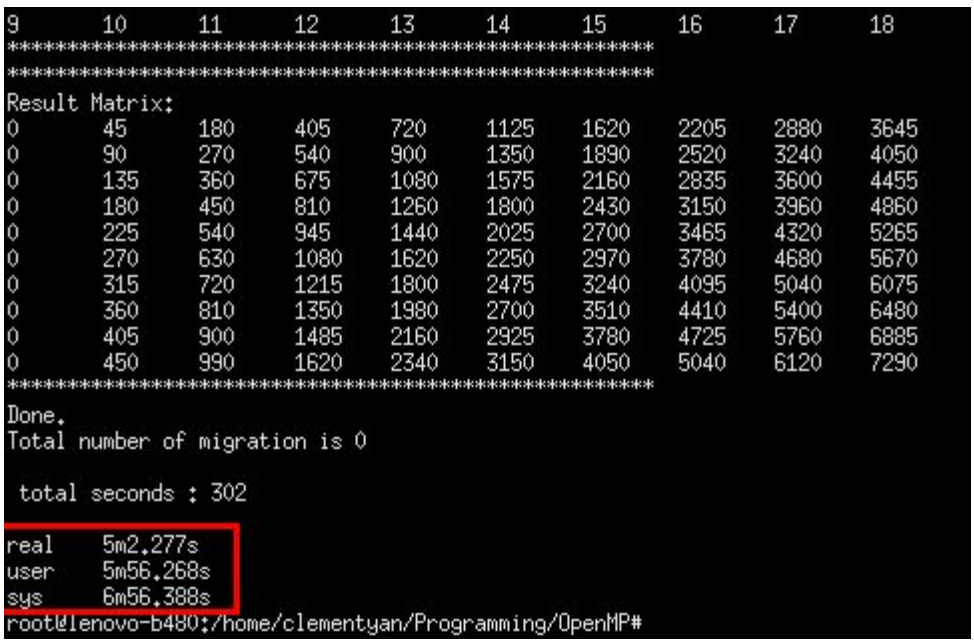
Default：

```
struct sched_param sp;
sp.sched_priority=sched_get_priority_max(SCHED_OTHER);
ret=sched_setscheduler(0,SCHED_OTHER,&sp);
```

```
top - 16:21:21 up 14:21, 14 users,  load average: 3.24, 3.26, 3.01
Tasks: 274 total,   1 running, 273 sleeping,   0 stopped,   0 zombie
%Cpu(s): 36.3 us, 35.9 sy,  0.0 ni, 27.7 id,  0.1 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  12085800 total,  7032400 used,  5053400 free,   187180 buffers
KiB Swap:   975868 total,        0 used,   975868 free.  1213056 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
22748 root      20   0   67332   3008   1320 S 287.6  0.0   3:57.37 DEFAULT
 5554 root      20   0  369940   7368   5796 S   2.7  0.1   0:07.95 udisksd
 1263 root      20   0  164780  30344  13020 S   1.3  0.3   7:18.83 Xorg
 1925 clement+  20   0 1458168 176472  63816 S   1.3  1.5  10:53.79 chromium
 3994 clement+  20   0 1493268 203924  84216 S   1.3  1.7  10:53.06 chromium
 1773 clement+  20   0 1426752 160012  63064 S   1.0  1.3  10:19.89 chromium
 1873 clement+  20   0 1445448 175968  71744 S   1.0  1.5   9:52.64 chromium
 1874 clement+  20   0 1464244 197116  69684 S   1.0  1.6  11:48.82 chromium
 3054 clement+  20   0 1491628 196716  75068 S   1.0  1.6  10:31.29 chromium
 1341 clement+  20   0  116572  20384  15600 S   0.7  0.2   1:19.86 icewm
 1419 clement+  20   0 3423828 385900 124864 S   0.7  3.2  18:30.75 chromium
 3690 clement+  20   0 1503356 218008  90080 S   0.7  1.8  11:52.51 chromium
 5548 clement+  20   0  870160 236520  43084 S   0.7  2.0   1:02.22 shutter
12819 clement+  20   0 1682280 380128 111472 S   0.7  3.1   7:14.72 chromium
 1322 clement+  20   0  369024  11544   6236 S   0.3  0.1   2:31.38 ibus-daemon
 1346 clement+  20   0  413604  35428  28780 S   0.3  0.3   0:44.08 ibus-ui-gt+
 1348 clement+  20   0  283896  12012  10744 S   0.3  0.1   0:27.35 ibus-x11
```



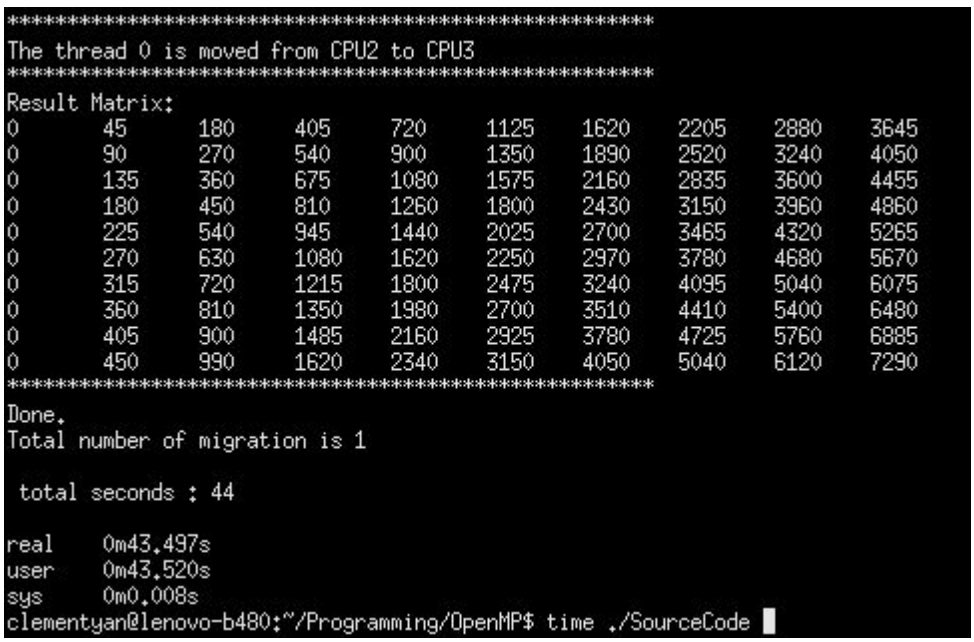


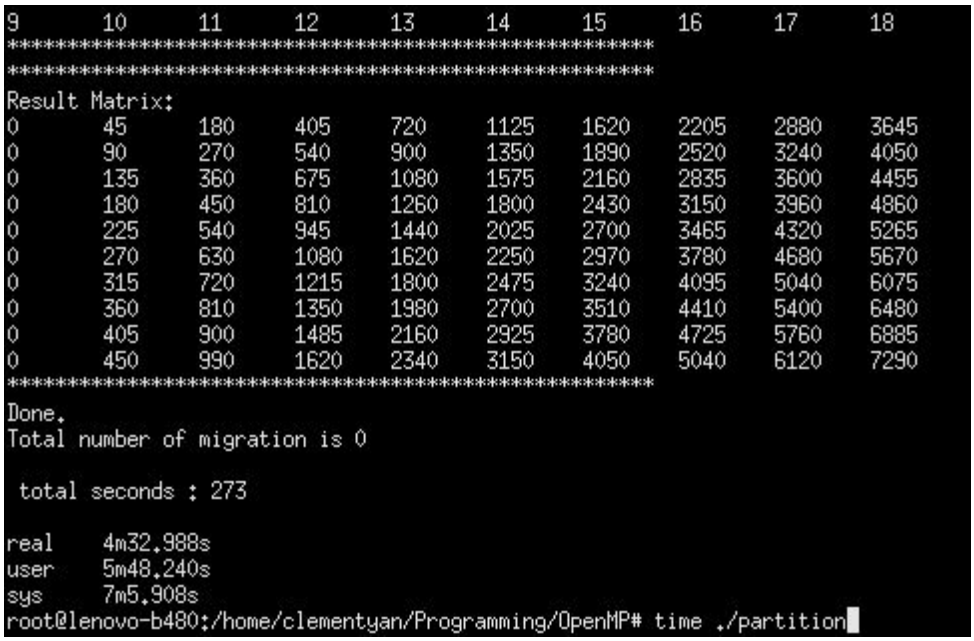
default的執行結果就會花較多的時間

show the scheduling states of tasks 10%

[Result. 20%]

Compare the response time of the program in three execution types(Serial, Global, Partition) 10%

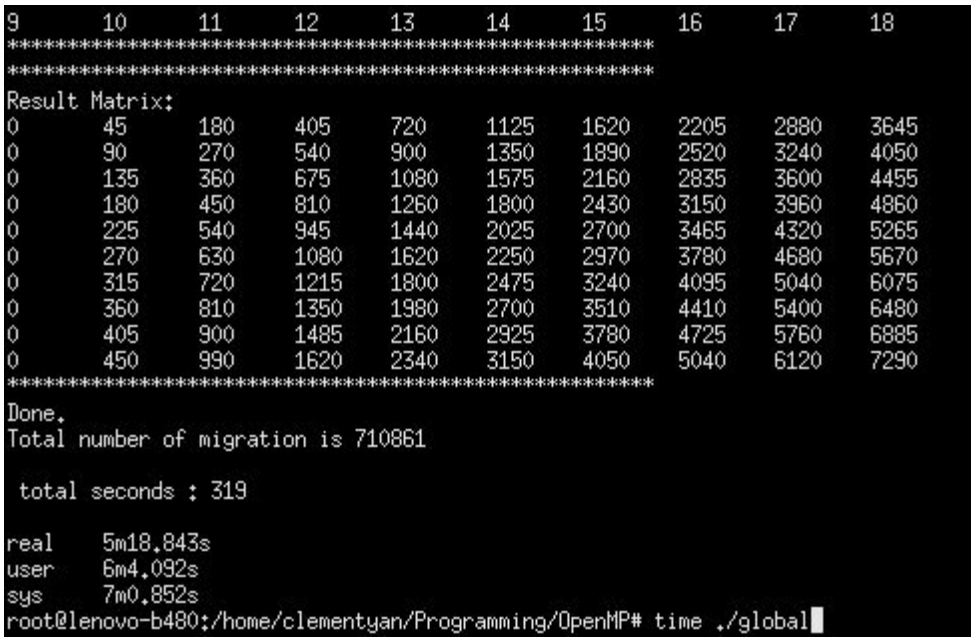




```
9       10      11      12      13      14      15      16      17      18
************************************************************
************************************************************
Result Matrix:
0       45      180     405     720     1125    1620    2205    2880    3645
0       90      270     540     900     1350    1890    2520    3240    4050
0       135     360     675     1080    1575    2160    2835    3600    4455
0       180     450     810     1260    1800    2430    3150    3960    4860
0       225     540     945     1440    2025    2700    3465    4320    5265
0       270     630     1080    1620    2250    2970    3780    4680    5670
0       315     720     1215    1800    2475    3240    4095    5040    6075
0       360     810     1350    1980    2700    3510    4410    5400    6480
0       405     900     1485    2160    2925    3780    4725    5760    6885
0       450     990     1620    2340    3150    4050    5040    6120    7290
************************************************************
Done.
Total number of migration is 0

 total seconds : 273

real    4m32.988s
user    5m48.240s
sys     7m5.908s
root@lenovo-b480:/home/clementyan/Programming/OpenMP# time ./partition
```



```
9       10      11      12      13      14      15      16      17      18
************************************************************
************************************************************
Result Matrix:
0       45      180     405     720     1125    1620    2205    2880    3645
0       90      270     540     900     1350    1890    2520    3240    4050
0       135     360     675     1080    1575    2160    2835    3600    4455
0       180     450     810     1260    1800    2430    3150    3960    4860
0       225     540     945     1440    2025    2700    3465    4320    5265
0       270     630     1080    1620    2250    2970    3780    4680    5670
0       315     720     1215    1800    2475    3240    4095    5040    6075
0       360     810     1350    1980    2700    3510    4410    5400    6480
0       405     900     1485    2160    2925    3780    4725    5760    6885
0       450     990     1620    2340    3150    4050    5040    6120    7290
************************************************************
Done.
Total number of migration is 710861

 total seconds : 319

real    5m18.843s
user    6m4.092s
sys     7m0.852s
root@lenovo-b480:/home/clementyan/Programming/OpenMP# time ./global
```

```
9       10      11      12      13      14      15      16      17      18
************************************************************
************************************************************
Result Matrix:
0       45      180     405     720     1125    1620    2205    2880    3645
0       90      270     540     900     1350    1890    2520    3240    4050
0       135     360     675     1080    1575    2160    2835    3600    4455
0       180     450     810     1260    1800    2430    3150    3960    4860
0       225     540     945     1440    2025    2700    3465    4320    5265
0       270     630     1080    1620    2250    2970    3780    4680    5670
0       315     720     1215    1800    2475    3240    4095    5040    6075
0       360     810     1350    1980    2700    3510    4410    5400    6480
0       405     900     1485    2160    2925    3780    4725    5760    6885
0       450     990     1620    2340    3150    4050    5040    6120    7290
************************************************************
Done.
Total number of migration is 0

 total seconds : 26

real    0m25.956s
user    1m40.112s
sys     0m0.000s
root@lenovo-b480:/home/clementyan/Programming/OpenMP# time ./partition_right
```

```
9       10      11      12      13      14      15      16      17      18
************************************************************
************************************************************
Result Matrix:
0       45      180     405     720     1125    1620    2205    2880    3645
0       90      270     540     900     1350    1890    2520    3240    4050
0       135     360     675     1080    1575    2160    2835    3600    4455
0       180     450     810     1260    1800    2430    3150    3960    4860
0       225     540     945     1440    2025    2700    3465    4320    5265
0       270     630     1080    1620    2250    2970    3780    4680    5670
0       315     720     1215    1800    2475    3240    4095    5040    6075
0       360     810     1350    1980    2700    3510    4410    5400    6480
0       405     900     1485    2160    2925    3780    4725    5760    6885
0       450     990     1620    2340    3150    4050    5040    6120    7290
************************************************************
Done.
Total number of migration is 6

 total seconds : 24

real    0m24.193s
user    1m32.844s
sys     0m0.000s
root@lenovo-b480:/home/clementyan/Programming/OpenMP# time ./global_right
```

從上面結果SourceCode是Serial其執行速度比partition與global來的快，主要原因是partition與global這兩個程式我刻意把平行處理寫在內層迴圈增加migration 次數，以證明說既使用parallel，但如果migration次數過多可能會比用serial執行所需的時間還來的久。
而後面兩個partition_right與global_right即是我把平行處理改會正確的外層for迴圈降低migration次數，可以看到執行速度皆比Serial來的快。

1.使用serial執行速度其實不一定會比用parallel的速度來的慢，因為parallel可能會造成很多migration overhead影響執行時間，而用serial因為都在同一個單一cpu way跑所以不會有migration overhead
2.從上面結果得知再相同fork出來的thread數量Global執行所需要的時間相對較partition久，原因如前所述因為partition綁定thread 與cpu way所以partition沒有migration overhead降低執行所需要的時間

3.但是partition雖然沒有migration overhead但執行所需的時間還是比serial還要久，解決方式就是讓process fork出來的thread數量<=cpu way的數量，Global也是如此，且要把平行處理寫在外層for迴圈

Describe your impression and the advantages/disadvantages of each algorithm 10%

1. 我發現使用float造成結果不正確如前面提到
2.我發現 process fork 出來的 thread 數量若超過 cpu way，將會提高migration的次數，原因前面有解釋
3.使用serial執行速度其實不一定會比用parallel的速度來的快，因為你可能會造成很多migration影響執行時間，而用serial因為都在同一個單一cpu way跑所以不會有migration overhead
4.使用partition可以綁定thread 與cpu way，可以避免migration，而Global會有migration現象
Serial：優點不會有migration overhead產生。缺點：只能使用單一cpu way來處理速度較慢
Global：優點：multithreading使處理速度提生。缺點：會有migration overhead產生
Partition：優點可以避免掉一些migration overhead產生，multithreading使處理速度提生

[Bonus. 10%](Choose one of following two)

Analyze the performance of three execution types

前面我有再執行結果說明一些分析與看法
Serial：使用單一個cpu way執行效能不會比較快，但不會有migration overhead產生
Global：使用parallel，但前面我有提到process fork出來的thread數量最好不要超過cpu way的數量，因為可能會增加migration的次數，甚至效能會比Serial來的慢，再者如果parallel程式沒有設計好也可能會產生過多的migration次數，如前所述將平行或處理放到內層for迴圈
Partition：使用Partition我覺的最大的幫助就是避免掉一些migration，使執行效能提昇

Analyze the performance of different schedulers (FIFO, RR, Default) in three execution types (Serial, Global, Partition)

Project submit

Submit deadline: 09 :00, May. 24, 2016
Submission : Moodle or M10307431@mail.ntust.edu.tw File name format : ESSD_Student ID_HW1.rar

※ Strictly prohibited copying !

ESSD _Student ID_HW1.rar must inculde the report and source code.

嚴禁抄襲，發生該類似情況者，一律以零分計算