HW2 document　　資訊三甲 10827139 黃云潔

1. 開發環境

- 開發平台：
  - OS : Windows11
  - CPU：Intel® Core™ i5-8265U
  - RAM : 8GB
- 使用開發環境：Visual Studio Code 1.61
- 使用的程式語言：C/C++

2. 實作方法與流程

流程

```
Input a file name (0: end of program):
input1
```

輸入檔案名稱，讀入檔案中需包含所選方法 1~6 和時間片段，及 process 的 ID、CPU Burst、Arrival Time、Priority 資訊，接著依照所選方法做排程，最終輸出該排程法得到的甘特圖，以及每個 process 的等待時間和往返時間。Process 中的 id 大於 9 時，改採用 A(10)、B(11)、…...替代。

實作方法

```
struct processInfo {
    int id;
    int cpuburst;
    int arrival;
    int priority;
};

vector<processInfo> pInfoList;
vector<string> outputList;
```

本次作業我的**儲存結構**使用 vector 存放每個 process 的資訊，outputList 存放 Gantt chart 字串。當選擇方法六時需執行方法一~五的排程，因此 outputList 會存不同方法的 Gantt chart。而實作方法過程中所需的 waiting queue 我也採用 vector，因使用 vector 也可以完成 queue 先進先出的功能。

在進實作方法的 function 前，我會先將 pInfoList **根據 arrival time (相同時用 process ID 由小到大)排序**，接著每個方法皆使用 **for 迴圈跑時間**，從第 0 秒開始，跑到 process 全都進入 waiting queue 且 waiting queue 已為空的且在 CPU 的 process 已執行完畢。進入迴圈後，比較 pInfoList 第一個 process 的 arrival time，若時間到了該 process 就進入 waiting queue，接著刪除 pInfoList 第一個 process。

- **方法一 FCFS**：依照 Arrival 的先後順序進行排序，若 Arrival Time 相同則依照 process ID 由小至大排序。

```
if ( time >= cpuEndTime ) {        // no process in CPU
    if ( waitingQueue.size() > 0 ) {       // have process in waiting queue
        cpuEndTime = time + waitingQueue.front().cpuburst;
        for ( int i = 0; i < waitingQueue.front().cpuburst; i++ )
            SetGanttName( waitingQueue.front().id, ganttChart );
        waitingQueue.erase( waitingQueue.begin() );
    } // if
    else
        ganttChart += '-';
} // if
```

waiting queue 的執行順序是 process 存進佇列的順序，直到前一個 process 完成，下一個等待的 process 才可以開始執行。

- **方法二 RR**：依照 Arrival 的先後順序進行排序，若 Arrival Time 相同則依照 process ID 由小至大排序，當執行的 process Timeout 時，該 process 若未執行結束則重新進佇列排隊；若提早結束，則讓直接下一個 process 開始執行。

```
if ( time >= cpuEndTime ) {        // no process in CPU
    if ( waitingQueue.size() > 0 ) {       // have process in waiting queue
        if ( pInCPU.cpuburst > 0 && pInCPU.id != -1 )
            waitingQueue.push_back( pInCPU );

        if ( waitingQueue.front().cpuburst >= time_slice )
            cpuEndTime = time + time_slice;
        else
            cpuEndTime = time + waitingQueue.front().cpuburst;

        pInCPU = waitingQueue.front();
        waitingQueue.erase( waitingQueue.begin() );
    } // if
} // if
pInCPU.cpuburst--;
```

waiting queue 的執行順序是 process 存進佇列的順序，當前一個 process 執行完一個 time slice 的時間，判斷該 process 若未做完(CPU burst > 0)，則將此 process 存入佇列，再讓下一個等待的 process 才開始執行。

- **方法三 SRTF**：依照剩餘 CPU Burst 由小到大進行排序，若剩餘 CPU Burst 相同則依照 Arrival Time 由小至大排序，Arrival Time 也相同則依照 process ID 由小至大排序

```
if ( waitingQueue.size() > 0 )
    SortByCPUBurst( waitingQueue );

if ( pInCPU.cpuburst <= 0 || pInCPU.id == -1 || pInCPU.cpuburst > waitingQueue.front().cpuburst ) {
    if ( waitingQueue.size() > 0 ) {       // have process in waiting queue
        if ( pInCPU.cpuburst > 0 && pInCPU.id != -1 )
            waitingQueue.push_back( pInCPU );
        pInCPU = waitingQueue.front();
        waitingQueue.erase( waitingQueue.begin() );
    } // if
} // if

pInCPU.cpuburst--;
```

waiting queue 的執行順序是 process 的 CPU Burst 由小到大排序，當在 CPU 中執行的 process 已經執行完畢，或 waiting queue 中有 CPU Burst

比執行中的 process 更小的 process，此時需換下一個 process 開始執行，未執行完的 process 則重新進佇列中根據剩餘 CPU Burst 排序。

- **方法四 PPRR**：依照 priority 由小到大排序，若有 priority 相同的 process 正在執行中，需等待其 Timeout；若佇列中有 process 的 priority 比正在執行的 process 更小，便會讓他奪取，若執行完的 process 未執行結束則排入佇列最後方，再根據 priority 由小到大排序。

```
if ( time >= cpuEndTime ) {        // no process in CPU
    if ( pInCPU.cpuburst > 0 && pInCPU.id != -1 ) {
        waitingQueue.push_back( pInCPU );
        SortByPriority( waitingQueue );
    } // if
    if ( waitingQueue.size() > 0 ) {
        if ( waitingQueue.front().cpuburst >= time_slice )
            cpuEndTime = time + time_slice;
        else
            cpuEndTime = time + waitingQueue.front().cpuburst;
        pInCPU = waitingQueue.front();
        waitingQueue.erase( waitingQueue.begin() );
    } // if
} // if
else if ( waitingQueue.size() > 0 && waitingQueue.front().priority < pInCPU.priority ) {
    if ( pInCPU.cpuburst > 0 && pInCPU.id != -1 ) {
        waitingQueue.push_back( pInCPU );
        SortByPriority( waitingQueue );
        if ( waitingQueue.front().cpuburst >= time_slice )
            cpuEndTime = time + time_slice;
        else
            cpuEndTime = time + waitingQueue.front().cpuburst;
        pInCPU = waitingQueue.front();
        waitingQueue.erase( waitingQueue.begin() );
    } // if
} // else if
pInCPU.cpuburst--;
```

waiting queue 的執行順序是根據 process 的 priority 由小到大排序，若 priority 相同則根據進入此佇列的順序排序。當前一個 process 執行完一個 time slice 的時間，或 waiting queue 中有 priority 比執行中的 process 更小的 process，此時需換下一個 process 開始執行。判斷若此 process 未執行完，則重新進入佇列中根據 priority 由小到大排序後，再讓下一個等待的 process 才開始執行。

- **方法五 HRRN**：依照反應時間比率由大到小進行排序，若反應時間比率相同則依照 Arrival Time 由小至大排序，Arrival Time 也相同則依照 process ID 由小至大排序。

```
if ( time >= cpuEndTime ) {        // no process in CPU
    if ( waitingQueue.size() > 0 ) {       // have process in waiting queue
        SortByResponseRatio( waitingQueue, time );
        cpuEndTime = time + waitingQueue.front().cpuburst;
        for ( int i = 0; i < waitingQueue.front().cpuburst; i++ )
            SetGanttName( waitingQueue.front().id, ganttChart );
        waitingQueue.erase( waitingQueue.begin() );
    } // if
    else
        ganttChart += '-';
} // if
```

反應時間比率計算：

$$\frac{(time - arrival\ time) + Waiting\ Time + CPU\ Burst\ Time}{CPU\ Burst\ Time}$$

waiting queue 的執行順序是根據 process 的反應時間比率由大到小排序，當前一個 process 執行完畢，計算 waiting queue 中所有 process 當下的反應時間比率並排序，取反應時間比率最大的 process 開始執行。

3. 不同排程法的比較

- **Average Waiting Time**

	FCFS	RR	SRTF	PPRR	HRRN
input1(time slice = 1)	14.33	18.4	8.07	14.7	11.6
input2(time slice = 3)	8.4	6.4	3	9.4	8.2
input3(time slice = 10)	6.67	11.67	6.67	12.5	6.67

- **Max Turnaround Time**

	FCFS	RR	SRTF	PPRR	HRRN
input1(time slice = 1)	26	45	57	56	29
input2(time slice = 3)	17	24	24	23	23
input3(time slice = 10)	45	55	45	60	45

- **FCFS**：平均等待時間**適中**，最大工作往返時間相對**最短**，因為使用先進先出的方式排程，不會有任何 process 會根據別的因素造成一直等待
- **RR**：平均等待時間**偏長**，最大工作往返時間相對**較長**，是相對公平的演算法，但會因 time slice 的選擇而造成效果不同，當 time slice 選擇太大，效果會同 FCFS；若 time slice 選擇太小，當有非常多個 process 需進行排程，每個人只用一點點就要重新排隊，會造成等待時間太長，效率差

Time slice	input1	input2	input3
1	18.4	6.6	13.17
3	16.27	6.4	13.67
5	16.27	7.2	12.5
8	14.33	8.6	14.67
10	14.33	10.2	11.67
15	14.33	8.4	9.17
20	14.33	8.4	11.67

- **SRTF**：平均等待時間**最短**，最大工作往返時間相對**較長**，因為此方法選擇剩餘 CPU 時間最短的 process 先執行，平均等待時間會縮短，但若遇到剩餘 CPU 時間很長的就會一直等到所有 process 結束，因此易造成有 process 等很久
- **PPRR**：平均等待時間**偏長**，最大工作往返時間相對**較長**，此方法可能會因 time slice 的選擇造成等待時間很長，也可能根據 priority 排程造成奪取，因此 priority 非優先的 process 一直輪不到，造成等待時間較長

- **HRRN**：平均等待時間**偏短**，最大工作往返時間相對**較短**，為了不使某些 process 因 CPU Burst 太長而導致等待時間很久，利用比較反應時間比率，此比率可遵守使 CPU Burst 越短的 process 優先處理，(例：兩 process 有相同 waiting time 但有不同的 CPU Burst)

	A		B
CPU Burst	3	<	5
等待時間	1		1
反應時間比率	1+3/3 = **1.33**	>	1+5/5 = **1.2**

同時使等待時間太長的優先處理，因此整體等待時間偏短

4. 結果與討論

將五個排程法分類，FCFS、HRRN 是屬於不可奪取的，RR、SRTF、PPRR 屬於可奪取的。RR 可能遇到時間片段選擇太大或太小的問題；SRTF 理論上可行且有最短的平均等待時間，但實務上並不可行，因為處理程序尚未執行前我們無法預估其處理時間；PPRR 可能會有優先等級較高的 process 一直進入，導致低優先等級的處理程序可能永遠無法被執行以致 Starvation。

以上歸納下來，若排程根據的優先順序與 arrival time 有關，如 FCFS 及 RR 是以 arrival time 排序、HRRN 以反應時間比率排序，皆不會造成 Starvation 的問題，而其他排程法若遇 Starvation，則可使用 aging 機制來解決。