PROJECT_2

F14051091 周易廷

Part A – Scheduler 實作

1. 操作說明:

甲、實作 Scheduler 的部分包含五支 C++程式，分別在五個對應名稱的不同資料夾中。這五支程式都是用 Visual Studio 撰寫，因此若要在 terminal 下編譯，需另外將標頭檔 #include "stdafx.h" 註解掉才能成功編譯。

乙、在 IDE 環境下，需要輸入 g++ 檔名，再輸入./a.out 才能執行

丙、另外，此程式在執行不同 data 的部分需要透過修改程式碼才能對不同 data 進行 Scheduling。

丁、執行完畢後，程式會將結果輸出成 csv 檔，以便做進一步分析。

戊、模擬的結果已經存在五個不同的資料夾，可以直接點開觀看。

	A	B	C	D	E	F	G	H	I	J
1	Context Switch 次數:	594	Context Switch Cost:	0	Average Waiting Time:	4630.73	Average TurnAround Time:	4632.63	Throughput:	0.526039
2										
3	id	turnaround time	waiting time	priority	burst time	arrive time	finished time			
4	1	1	0	1	1	1	2			
5	2	3	0	2	3	2	5			
6	3	3	1	4	2	4	7			
7	4	3	1	3	2	6	9			
8	5	4	3	10	1	6	10			
9	6	5	4	9	1	6	11			
10	7	38	34	4	4	6	44			
11	8	8	7	10	1	7	15			
12	9	9	8	1	1	7	16			
13	10	12	9	3	3	7	19			
14	11	13	11	4	2	8	21			
15	12	15	13	6	2	8	23			
16	13	16	15	4	1	8	24			
17	14	18	16	7	2	8	26			
18	15	20	18	10	2	8	28			
19	16	20	19	2	1	9	29			
20	17	21	19	10	2	10	31			
21	18	23	21	2	2	10	33			
22	19	22	21	3	1	12	34			
23	20	24	21	1	3	13	37			
24	21	25	24	10	1	13	38			
25	22	27	25	4	2	13	40			
26	23	29	26	1	3	14	43			

- Round-Robin Scheduling with context switch cost = 0 and time quantum = 3

2. 程式概念:

在程式設計的部分，五支程式所使用的 data structure 大同小異，都是使用 list 或 vector 來作為模擬的 ready queue，並且使用 array 紀錄讀到的 data 與輸出的結果。另外，他們的 main function 架構也幾乎相同，都會遵循一樣的流程(Read File → Simulating → Export Result)，差別只在模擬排程的 function 不同。

- FCFS function:

  使用 while 迴圈模擬時間，tu 為時間變數，並使用 arrival function 根據到達時間將新的 process 存入 Queue(用 list 代替)，在 Processor 的部分，利用定義好的 struct 作為模擬的 CPU 隨著 while 迴圈減少 burst time。在每次進入迴圈時先呼叫 arrival 更新 process，接著檢查 CPU burst time 是否為 0，如果為 0 且 Queue 內有 process 可以 run，則將 arrive time 最早的 process 複製進 CPU，並從 Queue 中 pop out。當完程的 process 數量與讀檔讀到的數量一致，跳出迴圈。

- SJN function:

  與 FCFS 相似，差別在將 Queue 中的 process 複製進 CPU 時選擇 burst time 最短的 process。

- Shortest-Remaining-Time-First function:

  與 SJN 相似，但是考慮到 Shortest-Remaining-Time-First 有 preemptive 特性與 context switch，不能只在 CPU 程式結束時檢查 Queue。因此在每次進入 while 迴圈時，除了判斷目前的 process 是否結束，也必須判斷 Queue 裡是否有 burst time 比當前 process 剩餘 burst time 更短的 process，如果有，則將目前 process 存入 Queue 尾端，將 burst time 更短的 process 複製進 CPU 執行。另外，考慮到 context switch 所花費的時間，必須在每次執行 context switch 後將變數 contextclock 調整到所花費時間的值，並隨時間逐步降到 0 為止，當 contextclock 不等於 0 時，CPU burst time 停止減少(不能執行)。

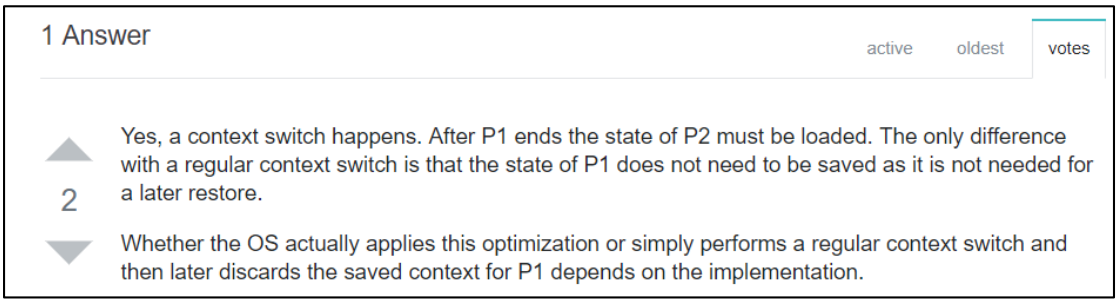
- Priority function:

  與 Shortest-Remaining-Time-First 相似，但是在考慮更換的時候，條件採用有最高 priority 的 process(priority 值最低)。

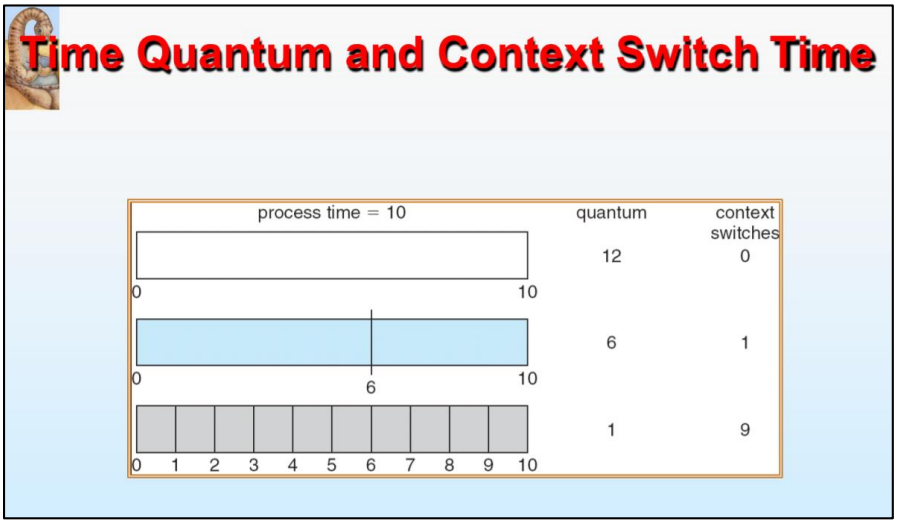
- Round-Robin function: 與 Prioriy 在 context switch 的部分相似，但是考慮到 Round-Robin 根據 time quantum 做 preempt 的特性，另外使用變數

counter 來計算目前 process 執行了多少 time unit。counter 一開始值為負無限大，當第一次有 process 複製進 CPU 後，counter 設為 0 開始計算，往後如果是因為程式執行結束才從 queue 中找程式出來，counter 設為 0，若是因為 time quantum 到了必須進行 Context Switch，則將 counter 設為 0-contextcost(Context Switch 花費時間)。另外在 process 因為超過 time quantum 而被 preempted 的時候，原來 process 會被放入 Queue 的尾端，而最前端的 process(最早到或重新排到)會被複製進 CPU。若同時有新 process 進入且發生 context switch，則新 process 加入到 Queue 先於 context switch 後排入的 process。

P.S. 在 Context Switch 發生的時機，根據 https://cs.stackexchange.com/questions/74049/does-a-context-switch-happen-when-a-process-has-terminated

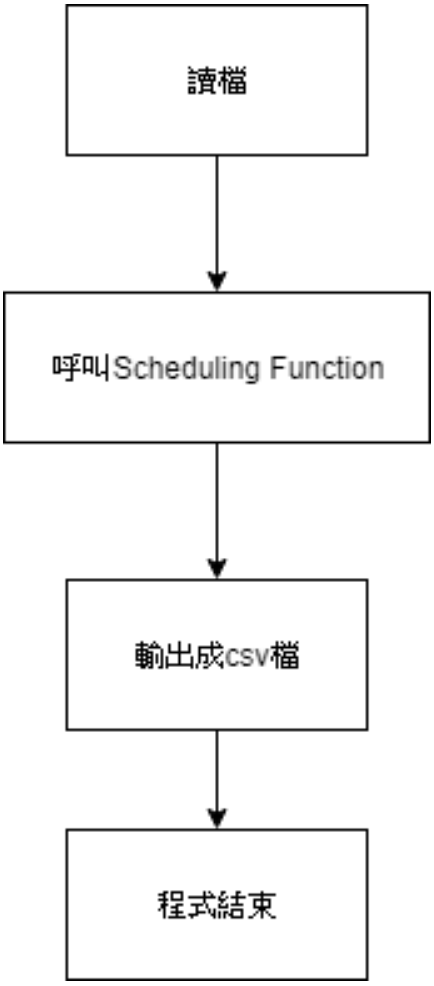


所述，在程式執行結束時，是否發生 Context Switch，會根據系統實作的方式決定，有些系統會不管 process 是否還在運行，都以 Context Switch 解決，有些則不會。因此，目前根據課堂講義示意圖

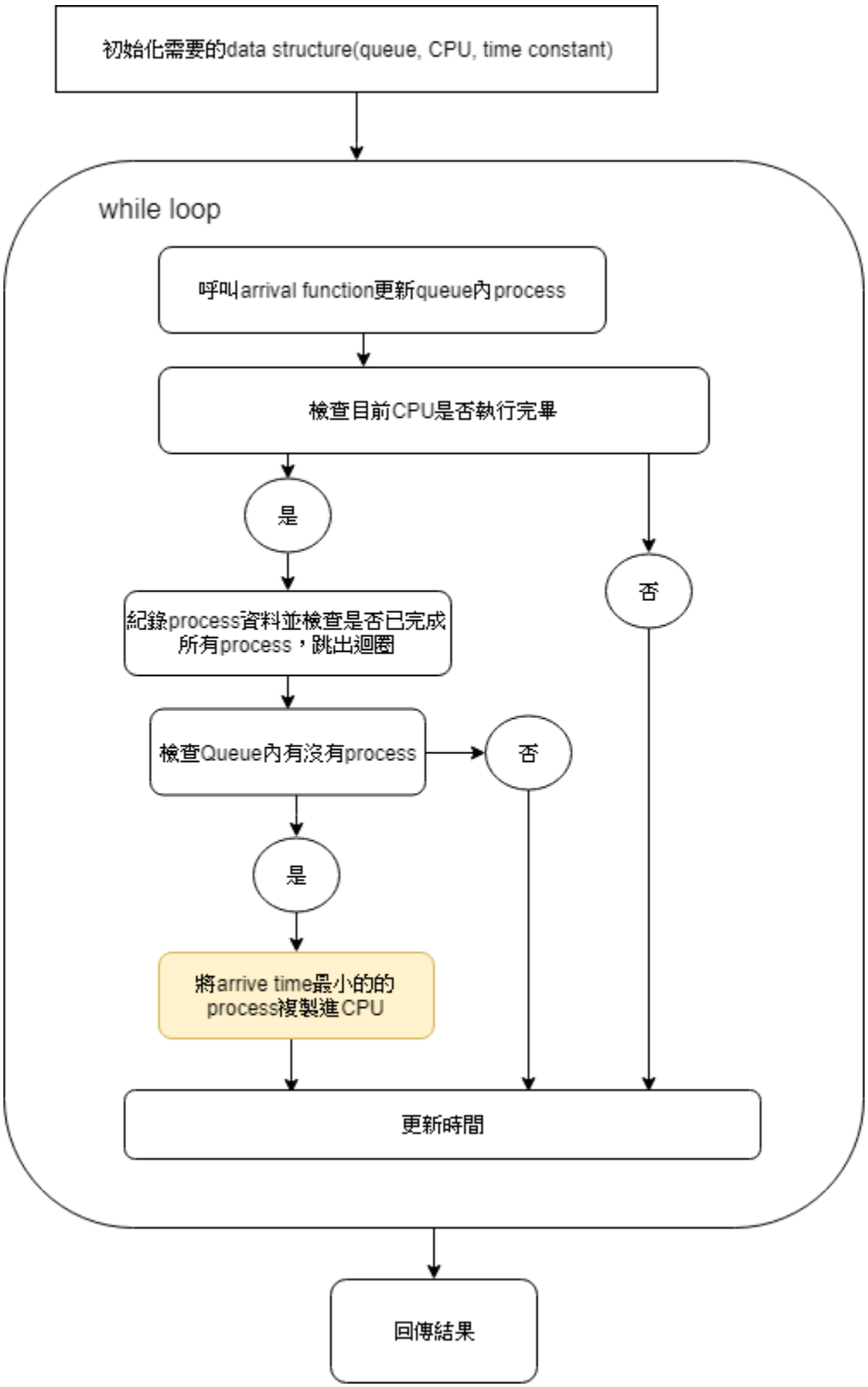


不考慮程式執行結束發生 Context Switch，因此 FCFS 與 SJN 都不會發生 Context Switch。

3. 程式流程圖:
   - Main function:

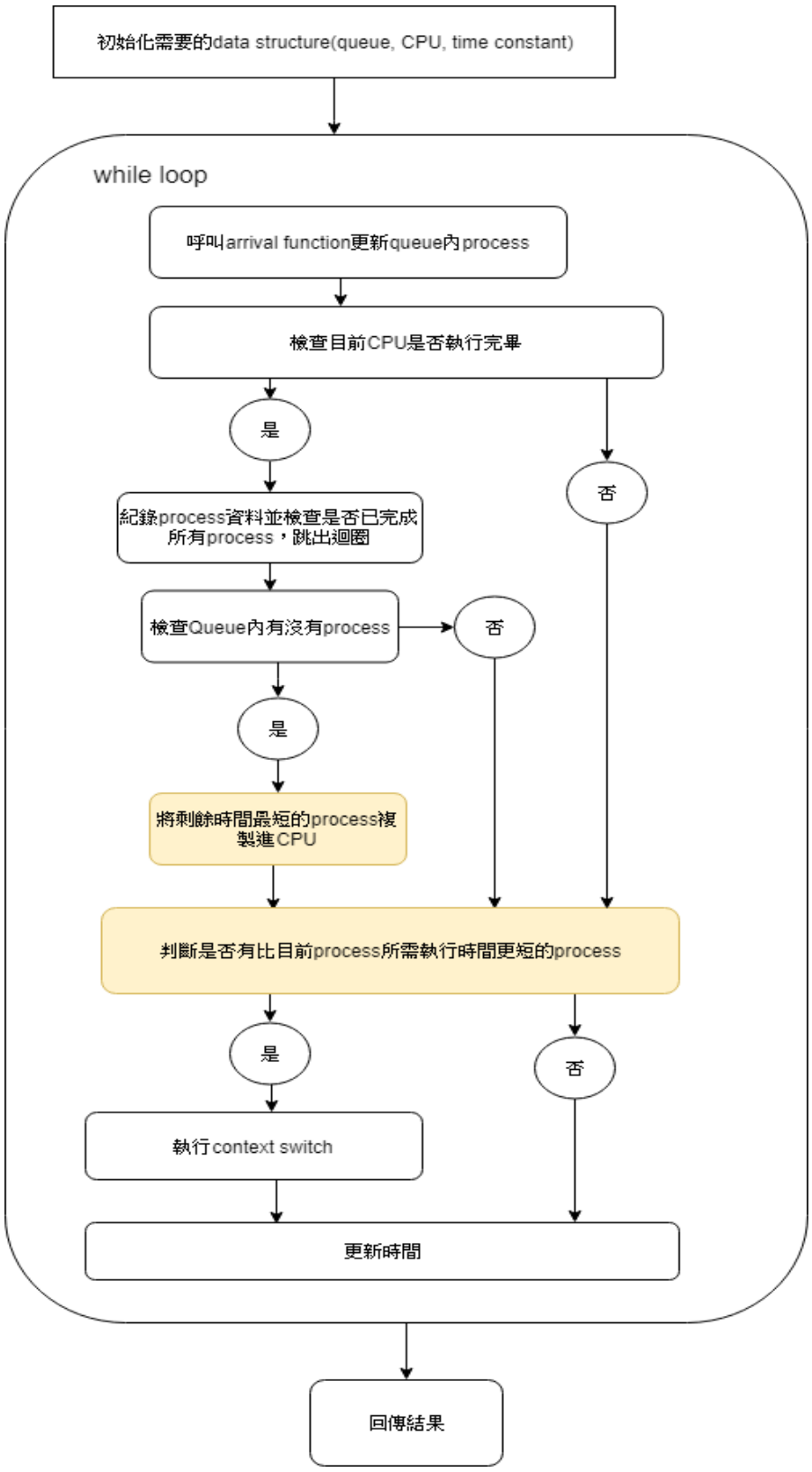


- FCFS function:



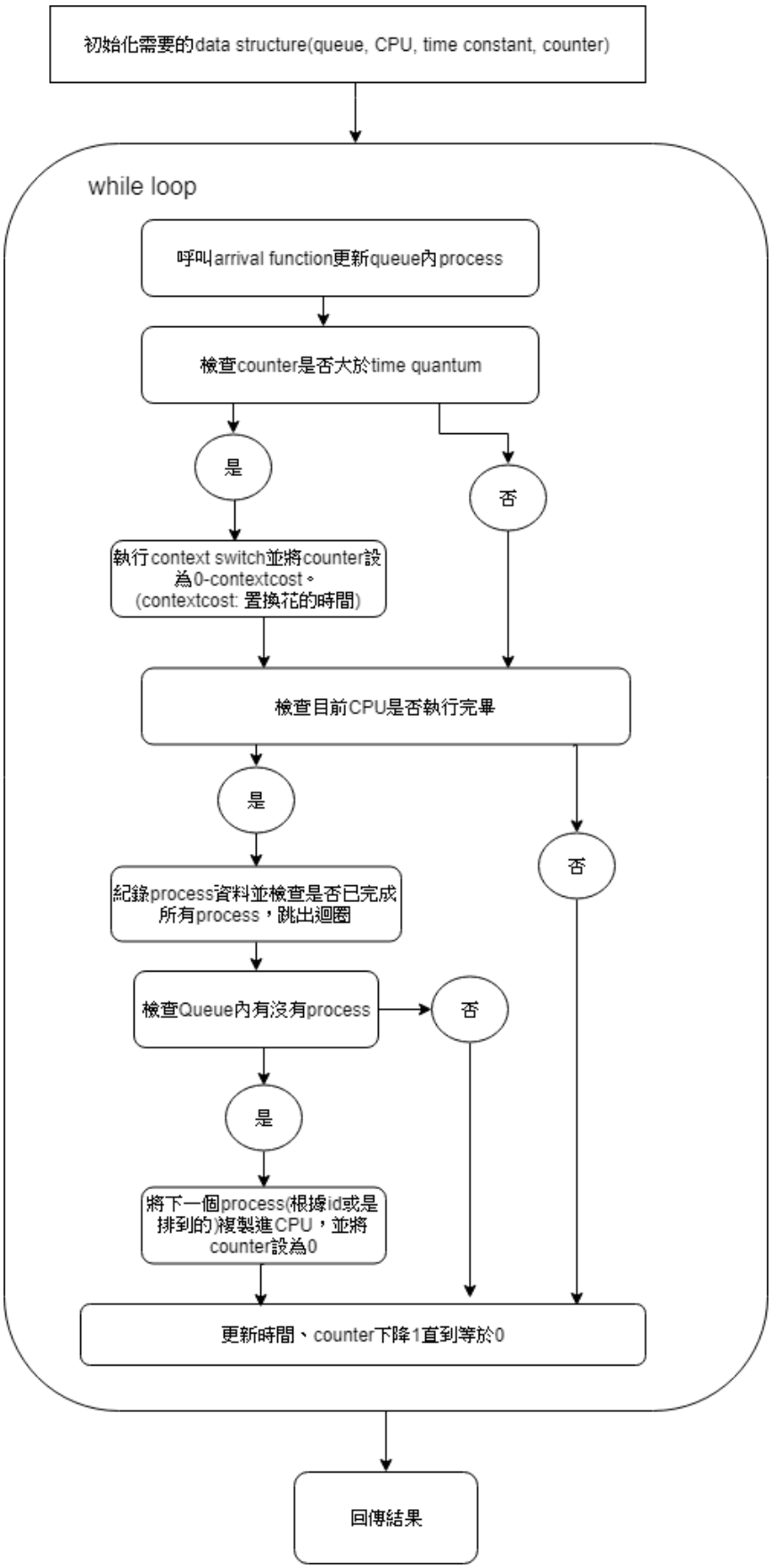
- SJN function:
  與 FCFS 差在橘色部分，判斷要 pop 哪個 process 時，根據 burst time 而不是 arrived time。

- Shortest-Remaining-Time-First function:



- Priority function:
  與 Shortest-Remaining-Time-First function 差別在於橘色部分是利用 priority 進行判斷。

- Round-Robin function:



Part B – Scheduler 效能分析

1. 程式結果(假設 Context Switch 所花費時間為 0)

Data1:

	FCFS	SJN	SRTF	Priority	RR(quantum = 3)
Average Waiting	4538.53	2707.63	2707.63	4530.12	4632.63
Average Turnaround	4540.43	2709.53	2709.53	4532.02	4630.73
Average Response	4538.53	2707.63	2707.63	3316.35	4365.76
Longest Waiting	9138	18652	18652	18754	13932
Longest Turnaround	9140	18658	18658	18755	13939
Longest Response	9138	18652	18652	15245	8779
Endtime	19010	19010	19010	19010	19010
Context Switch	0	0	0	9884	594
Throughput	0.526039	0.526039	0.526039	0.526039	0.526039

Data2:

	FCFS	SJN	Shortest-Remaining-Time-First	Priority	RR(time quantum = 15)
Average Waiting	184.001	119.245	118.376	247.306	241.291
Average Turnaround	199.044	134.288	133.419	262.349	256.334
Average Response	184.007	119.245	115.134	28.697	165.411
Longest Waiting	375	8421	8421	7976	660
Longest Turnaround	390	8446	8446	7987	677
Longest	375	8421	8421	478	332

Response					
Endtime	15065	15065	15065	15065	15065
Context Switch	0	0	154	1268	444
Throughput	0.066379	0.066379	0.066379	0.066379	0.066379

Data3:

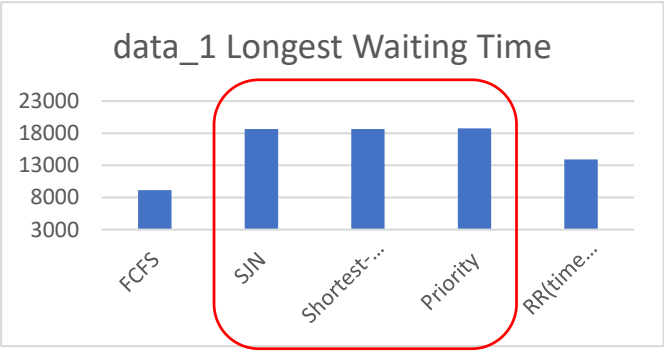
	FCFS	SJN	Shortest-Remaining-Time-First	Priority	RR(time quantum = 4)
Average Waiting	1987.41	1442.31	1441.76	1953.38	2536.17
Average Turnaround	1992.4	1447.3	1446.75	1958.37	2541.16
Average Response	1987.41	1442.31	1439.38	1547.36	1469.44
Longest Waiting	3988	4962	4692	4869	4634
Longest Turnaround	3992	4703	4703	4878	4643
Longest Response	3988	4962	4692	3908	2932
Endtime	4992	4992	4992	4992	4992
Context Switch	0	0	15	944	619
Throughput	0.200321	0.200321	0.200321	0.200321	0.200321

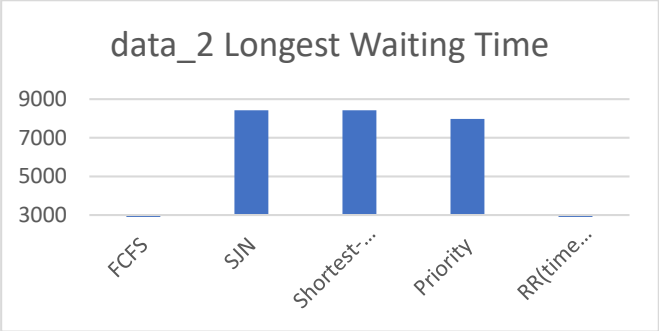
2. 效能分析

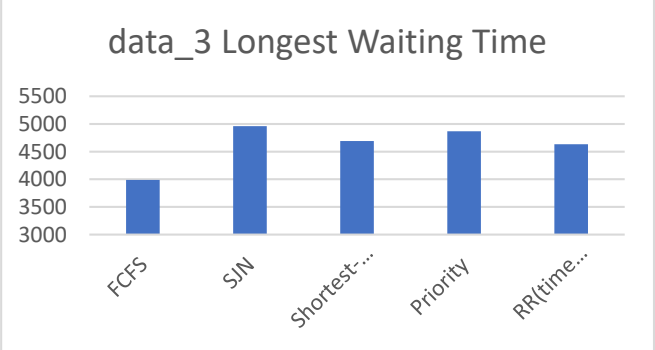
I. Starvation 問題:

從上面表格中，可以看到不管在哪一種 data，SJN、Shortest-Remaining-Time-First、Priority 都有著較高的 Longest Waiting Time，這是因為它們都有根據 priority 來排程的特性，與時間無關，因此容易產生 Starvation 問題。而 Round Robin 則介於 FCFS 與另外三者之間，因為 Round Robin 的時間分配上較平均，不會產生長時間無法執行的問題，儘管如此，在 data_3 中依然可以觀察到他的 Longest Waiting time 已經接近另外三者，考量到 data_3 中 process 出現較為頻繁，且執行時間不像 data_1 那樣短，容易造成大量積累的情況，導致某些需要被 context switch 較多次的 process 有著越來越多次

的排隊時間。同樣的現象也出現在 FCFS。

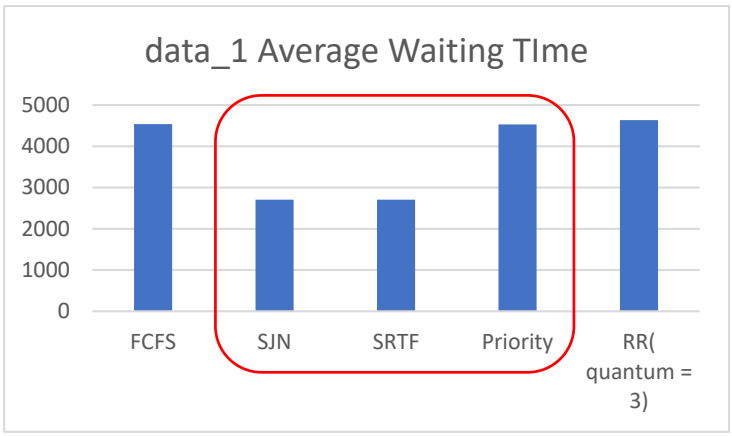


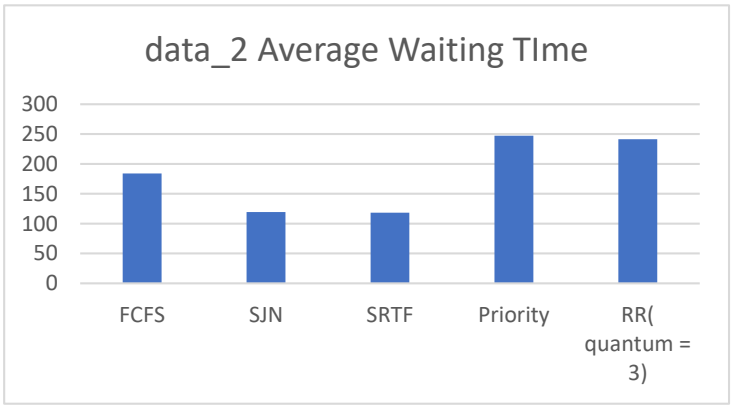


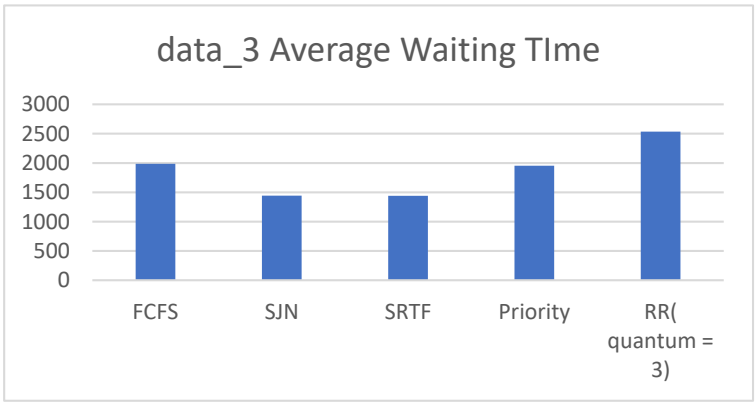


II. Average Waiting Time

在三種 data 的情況下，SJN 與 Shortest-Remaining-Time-First 都有著較好的 Average Waiting Time，這是因為在五種裡面，只有這兩種是以最快完成目前的 process 為目的，因此在相同情況下，它們的 Queue 中應該會有最少的 process 數量，也就使得總等待時間大幅降低；在 process 量越多越頻繁的情況下，SJN 與 Shortest-Remaining-Time-First 在 Average Time 上的優勢就會越明顯。以三種 data 為例，快又頻繁的 data_1 比頻繁的 data_3 明顯，而慢又資料量少的 data_2 最不明顯。

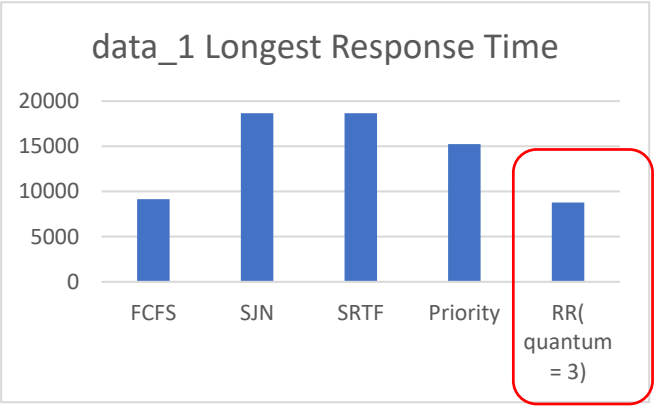

data_1 Average Waiting TIme
5000
4000
3000
2000
1000
0
FCFS
SJN
SRTF
Priority
RR(quantum = 3)

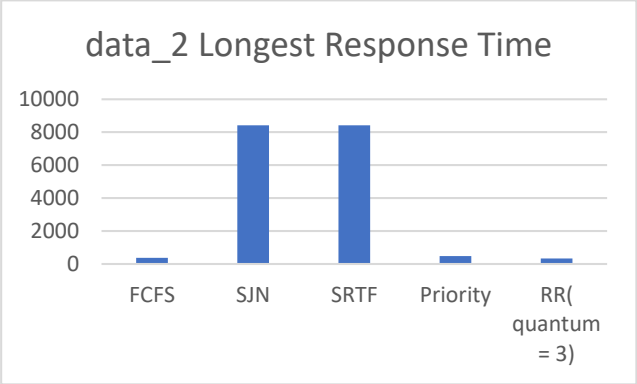

data_2 Average Waiting TIme
300
250
200
150
100
50
0
FCFS
SJN
SRTF
Priority
RR(quantum = 3)

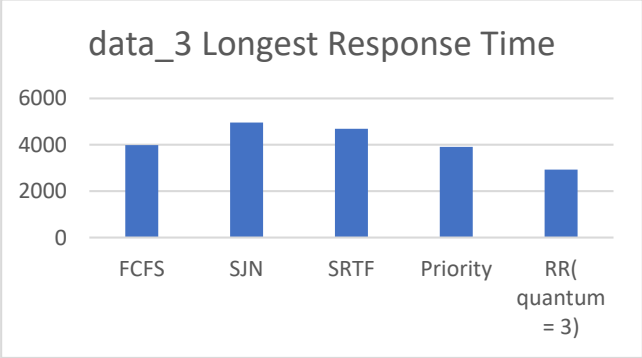

data_3 Average Waiting TIme
3000
2500
2000
1500
1000
500
0
FCFS
SJN
SRTF
Priority
RR(quantum = 3)

III. Response Time

在三種 data 中，Round Robin 都有最小的 Longest Response Time。但是在 Average Response Time 上就不一定，要根據實際情況而定。







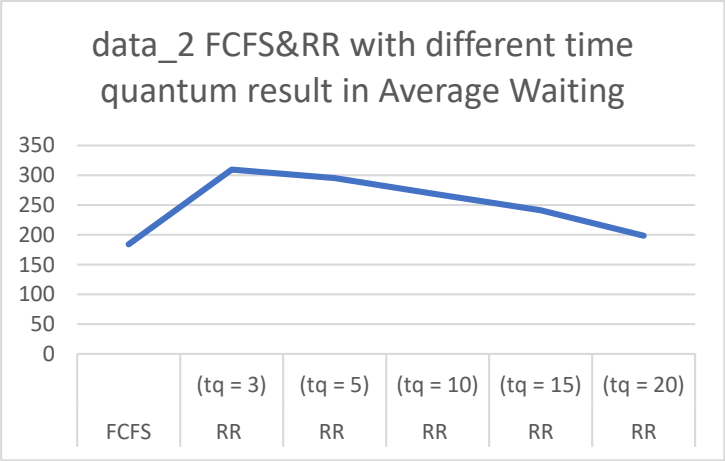
IV. Time Quantum 對 Round Robin 的影響

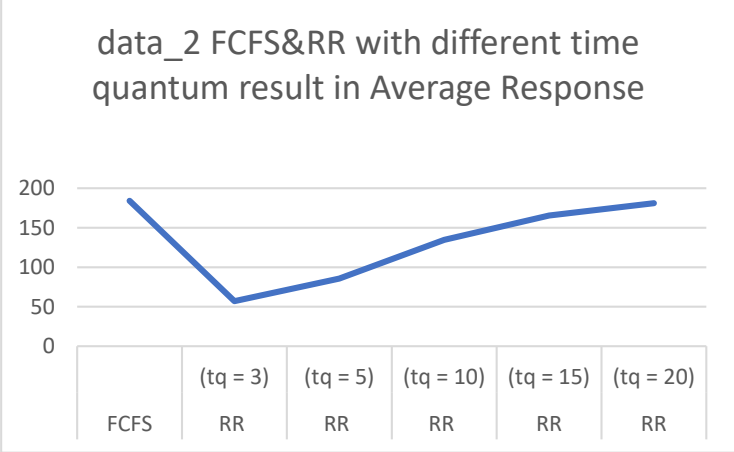
當 Time Quantum 較大時，Round Robin 會越來越傾向 FCFS，因此從圖中可以看出 Longest Waiting time 越來越接近 FCFS 的值，變得越來越小，其他值也越來越接近，可能原本 FCFS 的 Convey Effect 也會慢慢出現。

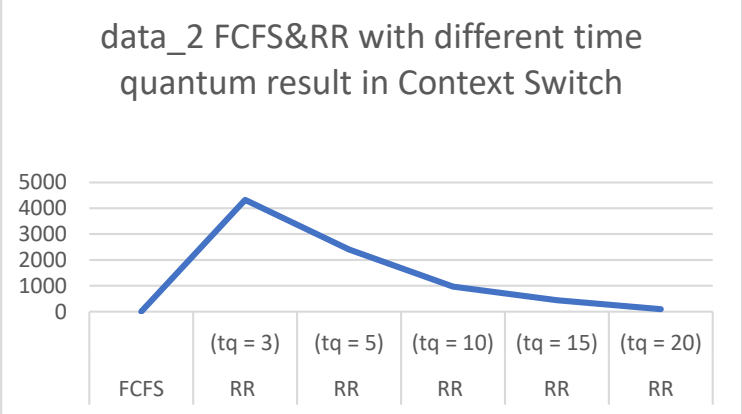
當 Time Quantum 太小時，會變得頻繁進行 Context Switch，此時許多有著較長 burst time 的 process 雖然等待時間會變快，但是也會需要排隊更多次。

Data_2

	FCFS	RR (tq = 3)	RR (tq = 5)	RR (tq = 10)	RR (tq = 15)	RR (tq = 20)
Average Waiting	184.001	309.35	295.233	267.663	241.291	198.573
Average Response	184.007	56.979	85.604	134.685	165.411	181.108
Context Switch	0	4327	2405	968	444	95





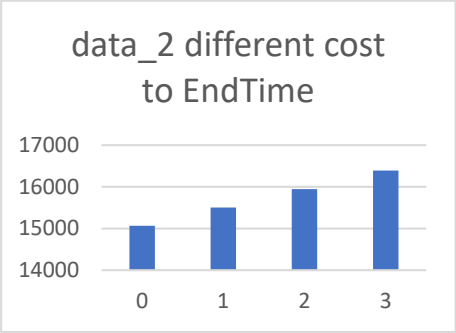


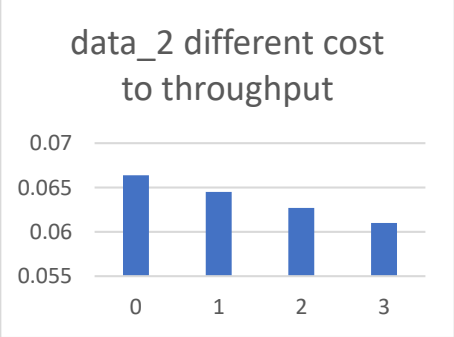
V. 存在 Context Switch cost 時，對 Round Robin 的影響

當存在 Context Switch cost 時，頻繁的 Context Switch 會大幅提高等待時間，也會使最終結束時間延長，而之所以 Context Switch = 0 的最終結束時間不等於 Context Switch = 1 時的最終結束時間扣掉 Context Switch = 0 時的 Context Switch 次數乘上 1，是因為在原本的排程中，本來就存在一些時刻，是 CPU 已經完成 process 且 Queue 中沒有 process 需要執行，所以在 Context Switch = 1 的情況下，CPU 閒置的時間會減少，因為許多花在 Context Switch 的時間也可以被用來等待新的 Process。

Data_2 Scheduled By Round Robin with time quantum = 15

Context Switch Cost	0	1	2	3
Endtime	15065	15504	15947	16391
Throughput	0.066379	0.0645	0.062708	0.061009





PART C-多處理器排班

現代電腦或手機中的 CPU 數量已經遠比以前多，而作業系統依照 CPU 的運作方式不同可以分成對稱式多處理器(SMP)與非對稱式多處理器(ASMP)。若多處理器中每個處理器都一樣能執行的功能也相同，即為對稱式處理器；反之，若不同 processor(或不同集合的 Processors)有著不同的能力與功能，且具有主從式結構，某一個 Processor 為 Master 其餘則為 Slaver，這種架構就是非對稱式多處理器(ASMP)

對稱多處理器(Symmetric multiprocessing,SMP):

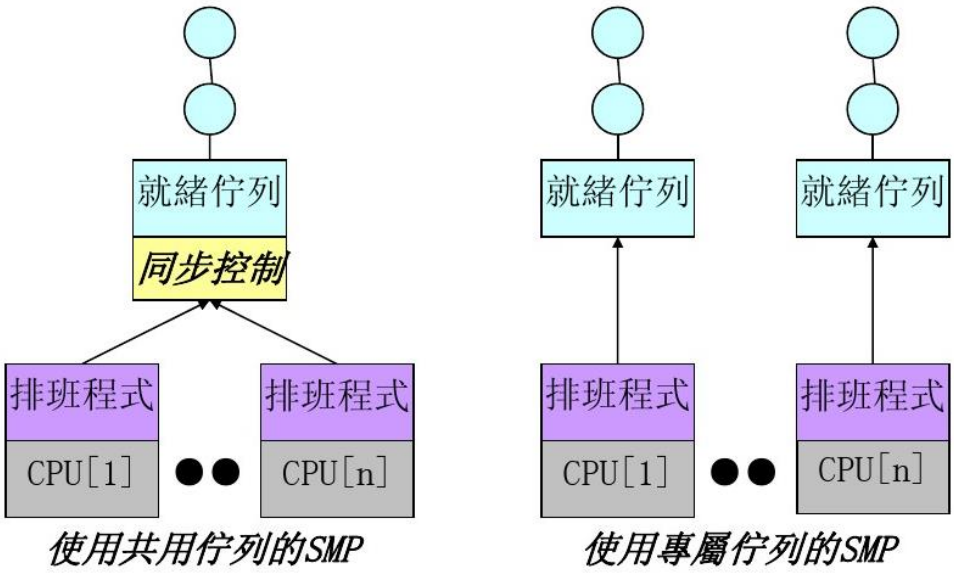
定義:

每個 Processor 的地位相同，功能也一樣，但需要共用記憶體。每個 processor 有自己的排班，並且執行自己的排班任務，當某一個 Processor 完成自己排班內的所有任務時，就會到就緒佇列去挑選行程，但有可能與其他的 Processsor 產生同步問題或兩個 Processsor 搶同一個任務執行。基本上決定 SMP 排班特性可

分為:有共用佇列與專屬佇列、負載平衡、與親和力(affinity)。

(1) **共用佇列與專屬佇列**:共用佇列就是每個 Processors 都共用同一個就緒佇列，當某個 Processor 執行完任務時便會到共用佇列去取得其他任務，因此當多個 processors 同時完成任務並取得其他任務時就有可能會發生同步問題。專屬佇列則是每個處理器都各自有自己的佇列，因此當處理器完成任務時只需到自己的主列去取得其他任務，現在大多數的 SMP 都是採用專屬佇列。

示意圖:



(2) 負載平衡(Load balance)

就如同字面上的意思，負載平衡的目的是為了讓所有行程行程平均分散到各個處理器中，不會讓有些處理器很忙碌而其他處理器在閒置的狀況發生，只要能盡可能達到負載平衡，就能讓系統的最大效能提升。通常為了達到負載平衡，有兩種方法:拉遷移與推遷移。拉遷移是將等待執行的行程拉到閒置的處理器中執行；而推遷移則是週期性的檢查處理器的負載，並將過於繁忙的處理器中的行程推給負擔較小的處理器去執行，拉遷移與推遷移可以同時並存。

(3) 親和性(Affinity):

由於當行程在某個處理器上執行時，處理器都會暫存該行程的資料，而當該行程在不同的處理器間轉移時，原本處理器上儲存的資料就會作廢，新的處理器又需要重新儲存該行程的資訊，這麼一來便很浪費系統的效能(如下圖所示)，因此系統會盡量讓某個行程在同一個(群)處理器中執行。其中又分為軟親和性(Soft affinity)與硬親和性(Hard affinity)。

軟親和性:系統會盡量讓某個行程待在同一個處理器中執行，但也有可能會轉移到其他處理器上。

硬親和性:只允許某個行程在同一個處理器或處理器集合內執行，不一定是在同一個處理器，但一定會在該集合內。

非對稱式多處理器(Asymmetric multiprocessing,ASMP):

定義:

每個處理器的功能不一樣，負責的工作也不相同。通常會有一個處理器當作 Master 負責協調與分配，而其他的處理器則為 Slaver，因此非對稱式多處理器具有主從關係。現在已經較少使用非對稱式處理器。

比較:

	SMP	ASMP
意義	每個行程在單一個處理器中執行	只有一個 master processor 在系統中執行任務
行程	行程從 ready queue 中取得	處理的行程只在 Master 中執行
架構	所有的處理器地位、功能均相同	每個處理器有不同的功能、地位(主從關係)
難易度	較複雜	較容易

多處理器排程方法介紹:

經常使用最長工作優先（LPT, Longest Processing-Time-first）演算法。其概念就是先將執行時間最長的任務進行排程，再將其餘的任務依執行時間長到短排到累計執行時間最短的處理器中，讓每個處理器盡可能達到負載平衡，以發揮系統最佳效能。

假設有 n 個任務需要執行(T1~Tn)，每個任務相對應的執行時間分別為(t1~tn)，並且有 m 個處理器(P1~Pm)

LPT 演算法的步驟如下:

步驟一:將各個任務所需要的執行時間由長到短排列

步驟二:把每個處理器目前的執行時間初始化為 0

步驟三:將任務序列中第一個任務指派給目前累計執行時間最短的處理器來執行

步驟四:累計該處理器的執行時間

步驟五:從任務序列終將該任務移除

步驟六:重複步驟 1~5 直到所有任務被分配執行完畢為止

步驟七:在所有的處理器中，累計完成任務時間最長者，當作最後完成時間。

在 LPT 演算法中，最後排程出來的時間不一定會是最佳解，也就是可能存在其他最短排程時間，但是 LPT 所需要花費的計算時間較其他可能產生最短時間的演算法來的短，而且也具有一定的時間優化能力。即使是使用 LPT 所產生的 worst case 也不會超過最佳解的 $\frac{4}{3}$ 倍。

LPT 實作:

程式碼:

```
# read and prepare n, m, and p
n = int(input("Number of jobs: "))
m = int(input("Number of machines: "))
pStr = input("Processing times: ")

p = pStr.split(' ')
for i in range(n):
  p[i] = int(p[i])

# machine loads and job assignment
loads = [0] * m
assignment = [0] * n

# in iteration j, assign job j to the least loaded machine
for j in range(n):

  # find the least loaded machine
  leastLoadedMachine = 0
  leastLoad = loads[0]
  for i in range(1, m):
    if loads[i] < leastLoad:
      leastLoadedMachine = i
      leastLoad = loads[i]

  # schedule a job
  loads[leastLoadedMachine] += p[j]
  assignment[j] = leastLoadedMachine + 1
```

```
# the result
print("Job assignment: " + str(assignment))
print("Machine loads: " + str(loads))
```

結果:

```
Number of jobs: 10
Number of machines: 3
Processing times: 8 7 6 5 5 4 4 3 3 3
8: [8, 0, 0]
7: [8, 7, 0]
6: [8, 7, 6]
5: [8, 7, 11]
5: [8, 12, 11]
4: [12, 12, 11]
4: [12, 12, 15]
3: [15, 12, 15]
3: [15, 15, 15]
3: [18, 15, 15]
Job assignment: [1, 2, 3, 3, 2, 1, 3, 1, 2, 1]
Machine loads: [18, 15, 15]
```

由於 LPT 演算法剛開始要先進行 N 個任務由時間長短排序一次，再走訪 M 個處理器找出最短累計時間的處理器並將在序列中執行時間最長的任務指派給累計執行時間最短的處理器，因此完成整個運算的執行時間需要 O(MN)。

PART C 參考資料(第 6 項為論文)

1. https://read01.com/zh-tw/AJmGPE.html#.XOfdv4gzZPY
2. https://ithelp.ithome.com.tw/articles/10203896
3. https://sls.weco.net/node/21325
4. https://theory.epfl.ch/osven/courses/Approx13/Notes/lecture2.pdf
5. http://ohlandl.ipv7.net/CPU/ASMP_SMP.html
6. https://www.sciencedirect.com/science/article/pii/0166218X88900790