一、開發環境

- Windows10 + VScode + TDM-GCC

二、實作方法和流程

- FCFS:

  先對原資料按照 arrival 由小到大排序，若相同則 id 小的先。接著塞進 unused_q，並用迴圈模擬時間流逝。每到一個 time spot 就讀入抵達的資料到 waiting_q，若此時並非閒置且沒有未執行完的 process，就從 waiting_q 中抓出第一個 process 執行；若有正在執行的就繼續執行到結束，若閒置且沒有抵達的 process 就跳下一個 time spot。

- RR:

  先對原資料按照 arrival 由小到大排序，若相同則 id 小的先。接著，初始化每個 process 對應的 std::pair<int, int> record_time，first 為此 process 上一次的執行 time spot，second 為用於累加的 waiting time。並且有 time_counter 對每個 process 進行監控。

  同樣用迴圈模擬時間流逝，並且根據每個時間點先將 unused_q 中隊應的資料塞進 waiting_q；再來若上一回的 process 還沒做完但到了 time slice 限制的時間，就塞進 waiting_q 並重設 time_counter。最後，若此時間點非閒置，就將 waiting_q 中的第一個 process 拿出來執行。若此 process 執行完，就重設 time_counter。

- SJF:

  先依 arrival time 由小到大排序，若相同則照 CPU Burst 由小到大排序，若再相同則照 id 由小到大排序。同樣始用迴圈模擬時間流逝，並按照時間點從 unused_q 中拉資料到 waiting_q。接著，若非閒置，則對 waiting_q 先按照 CPU Burst 由小到大排序，若相同則照 arrival time 由小到大排序，若再相同則照 id 由小到大排序。最後，將 waiting_q 的第一個 process 拔出執行，其餘同樣直到結束。

- SRTF:

  先依 arrival time 由小到大排序，若相同則照 CPU Burst 由小到大排序，若再相同則照 id 由小到大排序。同樣始用迴圈模擬時間流逝，並按照時間點從 unused_q 中拉資料到 waiting_q。再來與 SJF 不同的地方就是，SJF 會在非閒置時對 waiting_q 排序並抓第一個 process，但 SRTF 每次一定會對 waiting_q 先按照 CPU Burst 由小到大排序，若相同則照 arrival time 由小到大排序，若再相同則照 id 由小到大排序。並且在非閒置時抓第一個出來執行，其餘與 SJF 相同。

- HRRN:

  先對原資料按照 arrival 由小到大排序，若相同則 id 小的先。接著塞進 unused_q，並用迴圈模擬時間流逝。每到一個 time spot 就讀入抵達的資料到 waiting_q，並且與 FCFS 不同的地方是在有 process 結束時對所有 waiting_q 中的 process 計算 response ratio，並按照以下規則排序：response ratio 由大到小排序，若相同再按照 arruval time 由小到大排序，若再相同則照 id 由小到大排序。並且排序之後選擇 waiting_q 中的第一個 process 等到下一個 time spot 時執行。

- PPRR:

  先對原資料按照 arrival 由小到大排序，若相同則 priority 小的先，若再相同則 id 由小到大排序。接著，初始化每個 process 對應的 std::pair<int, int> record_time，first 為此 process 上一次的執行 time spot，second 為用於累加的 waiting time。並且與 RR 一樣有 time_counter 對每個 process 進行監控。

  接著，一樣用迴圈模擬時間流逝，並在每個時間點將 unused_q 中對應的資料按照以下規則插入 waiting_q 以不破壞原有的順序：priority 小的優先，若相同則 arrival time 小的先。

  再來，判斷要選擇哪一個 process。若 waiting_q 是空的，且手頭上沒有未執行完的 process 即為閒置；否則，若手頭上沒有未執行完的 process，就從 waiting_q 中取第一個執行；再不然就會有未執行完的 process，當 waiting_q 不為空時，若 waiting_q 中第一個 process 的 priority 比較大就重設 time_counter 並繼續執行未執行完的 process；若目前的 priority 較小，或是相同 priority 但已達 time slice 上限時，就重設 time_counter，並將目前未執行完的 process 插入到 waiting_q 中 priority 比目前 process 的 priority 大的最近一個 process 之前。例如(id, priority)：waiting_q = {(1, 3), (4, 5), (5, 5), (9, 11)}，current process = (293, 5)，就會插到(9, 11)之前（若不存在相同 priority 的 process 一樣會插到(9, 11)之前）。再來，從 waiting_q 中抓第一個 process 執行。如果手頭上有未執行完的 process，並且 time_counter 也未到達 time slice 的上限，且 waiting_q 非空時，一樣會照前述做法選取 process。最其餘與 RR 相同。

- 資料結構：

  i. struct Process: 除了 id, cpu_burst, arrival_time, priority 這四個提供的資料外，還包含要輸出的 waiting_time, turnaround_time，以及執行計算中使用的 finish_time,

remain_time, response_ratio。

ii. struct Data: 用於儲存讀入檔案的類別，包含 method, time_slice，以及包含 Process 的動態陣列 processes。

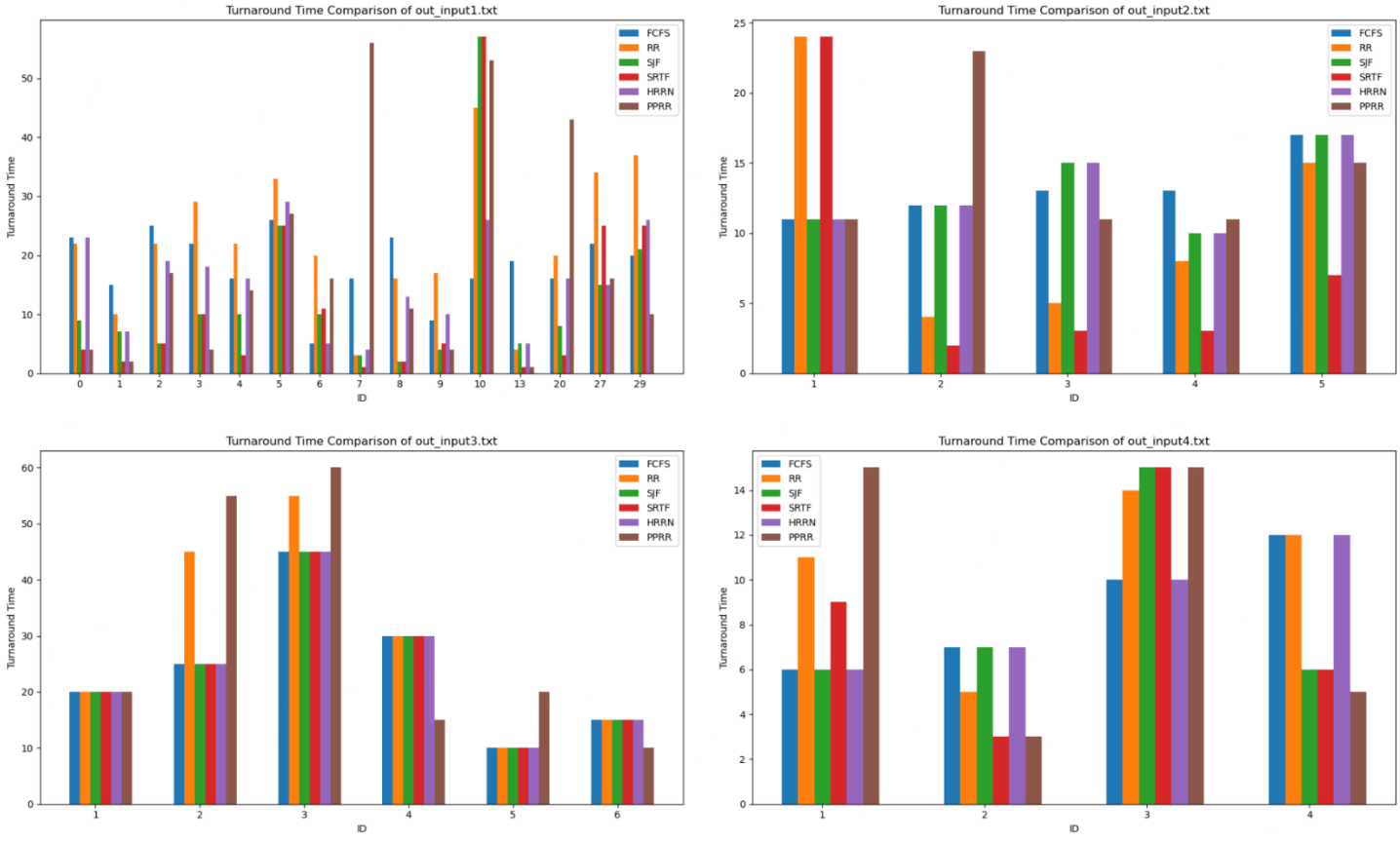
iii. class Method::struct Result: 用於儲存即輸出的結構，有包含 Process 的動態陣列 results 儲存結果，Gantt_chart 用於保存甘特圖，以及 method_name 用於寫檔。

三、不同排程法比較

- 平均等待時間(四捨五入到小數點後第二位)：

	FCFS	RR	SJF	SRTF	HRRN	PPRR
input1	14.33	18.4	8.67	8.07	11.6	14.67
input2	8.4	6.4	8.2	3	8.2	9.4
input3	6.67	11.67	6.67	6.67	6.67	12.5
input4	3.75	5.5	3.5	3.25	3.75	4.5

- 工作往返時間：



四、結果與討論

- 不同方法的效能結果，是否有跡可循？

  透過比較平均等待時間，我們可以粗略比較：

  SRTF<SJF<HRRN<FCFS<PPRR<RR。

  對於 FCFS，由於後面來的 process 必須等執行中的 process 做完，因

此平均等待時間會很長。而對 RR 而言，它完全取決於 time slice 的大小，在 input1 中 time slice = 1，input3 中 time slice = 10，而當 time slice 太小會因為頻率過高的 context switch 導致效率較差，太大會如同 FCFS 導致等太久，因此可以看到這兩個相對 input2(time slice = 3), input4(time slice = 2)來說，平均等待時間是很長的。PPRR 在 RR 的基礎上加上了優先度，但會有 process 餓死，如果要改善可以使用 dynamic priority。SJF 與 SRTF 都會不斷尋找 CPU Burst 最短的 process，但 SRTF 因為可奪取所以尋找的次數會更多，平均等待時間也會比 SJF 少一些。HRRN 綜合考慮了 waiting time 以及 CPU Burst，隨著等待時間愈久，ratio 愈大，繼而避免了 process 餓死的問題。不過相對地，等待時間會變長。

- 實際運用上，應如何挑選排程機制？

  挑選排程法時，應該考慮：

  1. process 是如何組織的？
  2. 資源如何利用？
  3. process 是如何進入的？
  4. 資料結構與演算法的設計？
  5. 是要效率優先，還是反應時間優先，或是其它指標？
  6. 是要 process 間交互處理還是批次處理？

等等因素，但主要還是要根據 task 的性質設計與選擇。例如在使用無人機即時連線戰場畫面與傳遞指令時，必須以最快的反應時間為優先，若晚了幾秒鐘友軍可能就被炸死了；而在進行 multi-processing 的平行運算時，就不能選擇會導致 process 餓死的排程法。所以，實際情況必須根據 task 的多項性質判斷，沒有最好的，只有最符合需求的。