</

作業系統排程模擬

} /> [

/>

職四訊三甲 9B117023
張庭睿

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1

</ 目錄

{01}
使用語言及模組

{02}
行程表轉換甘特圖&PCB內容

{03}
SJF甘特圖及中斷模擬

{04}
FCFS、RR、SJF、SRTF
排程模擬

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ 使用語言及模組

Python

Python 作為一款較簡易入門的語言，雖然比較難直接模擬電腦底層的運行，且執行效率較慢，但能夠簡單地制定類別及匯入模組。

其餘模組:

Random: 隨機產生PCB當中尚未要模擬操作的資料
Time: 模擬程序運行的時間，基本上以一秒為主
Deque from collection: 排程中，RR會使用到deque，故使用該模組
Pyinstaller: python要產生執行檔所需模組。

使用的GUI模組:Matplotlib

嚴謹來說，Matplotlib是一個分析實驗結果用的2D圖形繪圖模組，在這裡使用來表示排程後的甘特圖。

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ 行程定義

由於尚未構築完整的底層系統，僅模擬行程之PCB及運行的狀況，故除PID、到達時間、執行時間、剩餘時間外，其餘數值為模擬系統控制或隨機產生。

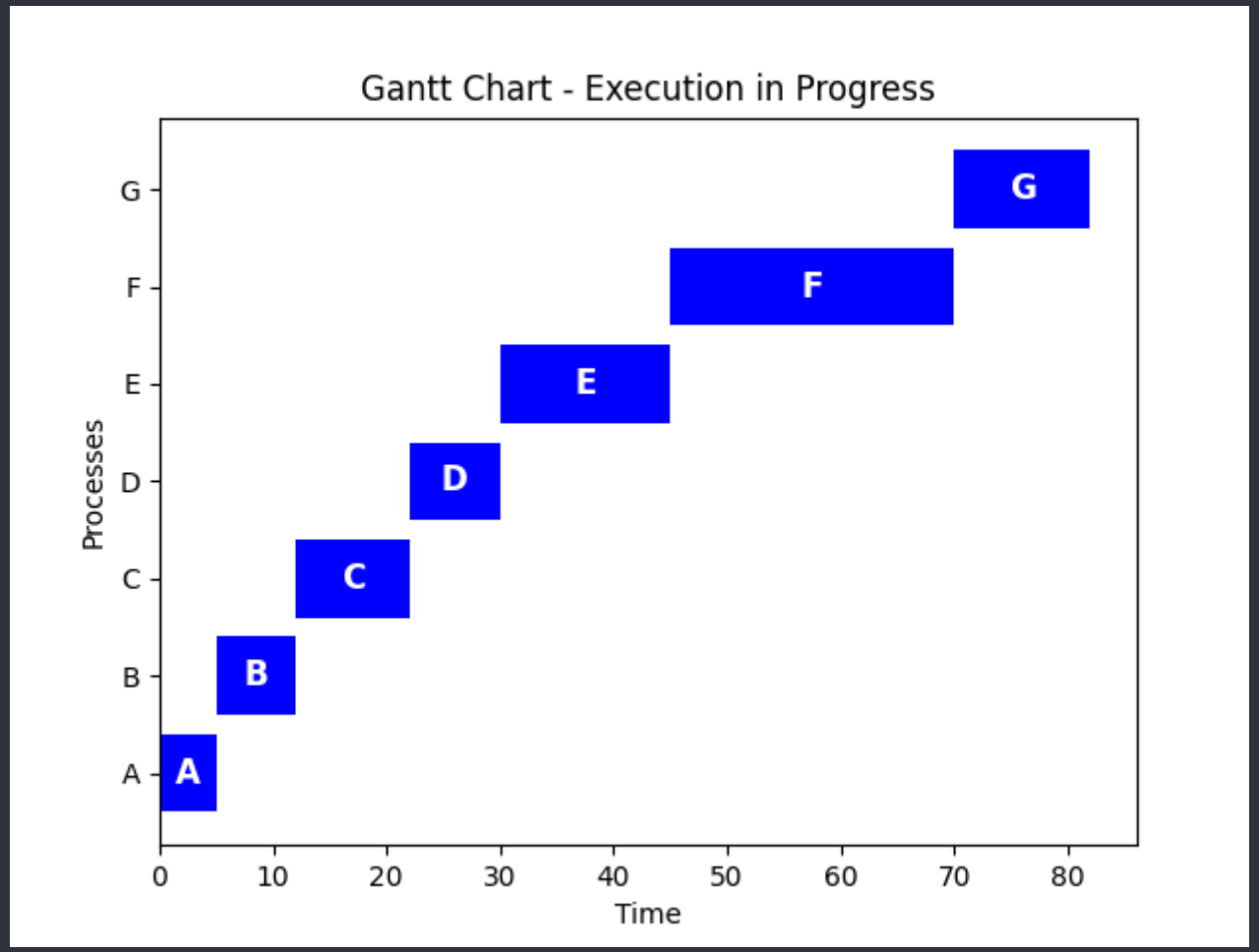
剩餘時間一開始設定為執行時間，並在每個系統週期後減去，供SRTF作為判斷依據。

行程Class

```python
# 定義 Process 類別，代表一個行程
class Process:
    def __init__(self, pid, arrival_time, burst_time):
        self.pid = pid  # 行程 ID
        self.arrival_time = arrival_time  # 到達時間
        self.burst_time = burst_time  # 執行時間
        self.remaining_time = burst_time  # 剩餘時間（供 SRT 使用）
        self.waiting_time = 0  # 等待時間
        self.turnaround_time = 0  # 周轉時間
        self.completion_time = None  # 完成時間
        self.state = "Ready"  # 行程狀態
        self.program_counter = 0  # 程式計數器
        self.registers = {f'R{i}': random.randint(0, 100) for i in range(4)}  # 模擬暫存器
        self.memory_limit = random.randint(100, 500)  # 記憶體限制
        self.open_files = [f'file_{pid}_{i}.txt' for i in range(random.randint(1, 3))]  # 已開啟檔案表
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</行程表轉換甘特圖&PCB內容



甘特圖

以FCFS表示該行程之甘特圖

PCB

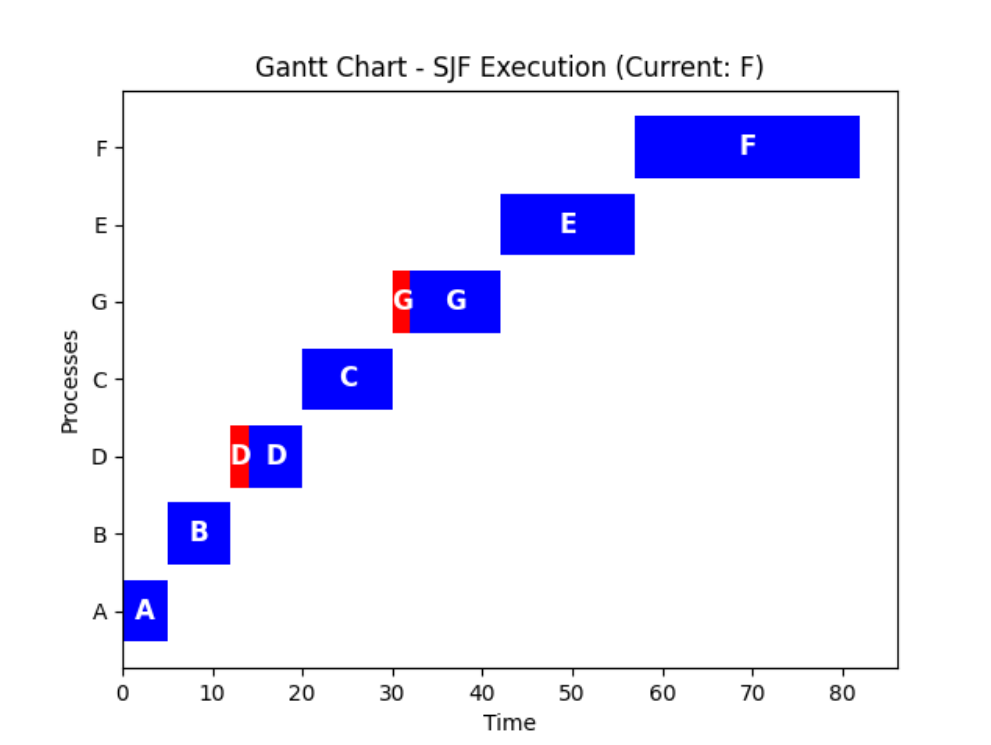
包含PID、PC、暫存器、記憶體、開啟檔案等

```
PCB for Process A:
  State: Running
  Process ID: A
  Program Counter: 2782
  Registers: {'R0': 39, 'R1': 11, 'R2': 84, 'R3': 72}
  Memory Limit: 400
  Open Files: ['file_A_0.txt', 'file_A_1.txt', 'file_A_2.txt']

PCB for Process A:
  State: Terminated
  Process ID: A
  Program Counter: 2782
  Registers: {'R0': 39, 'R1': 11, 'R2': 84, 'R3': 72}
  Memory Limit: 400
  Open Files: ['file_A_0.txt', 'file_A_1.txt', 'file_A_2.txt']
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ SJF甘特圖及中斷模擬



SJF: 最短程序優先

當運行時，以最短運行時間為優先進行排序。

```
while self.processes and self.processes[0].arrival_time <= self.time_counter:
    self.ready_queue.append(self.processes.pop(0))
    self.ready_queue.sort(key=lambda p: p.burst_time)
```

中斷模擬:

模擬將會被中斷之後繼續執行的程序

```
self.interrupted_processes = {"D", "G"}  # D 和 G 會被中斷
self.suspended_processes = set()  # 記錄已被中斷的行程
```

甘特圖:

被中斷的程序會先呈現紅色中斷狀態，中斷程序結束後，繼續執行，程序運行中會產生相對應的PCB表。

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

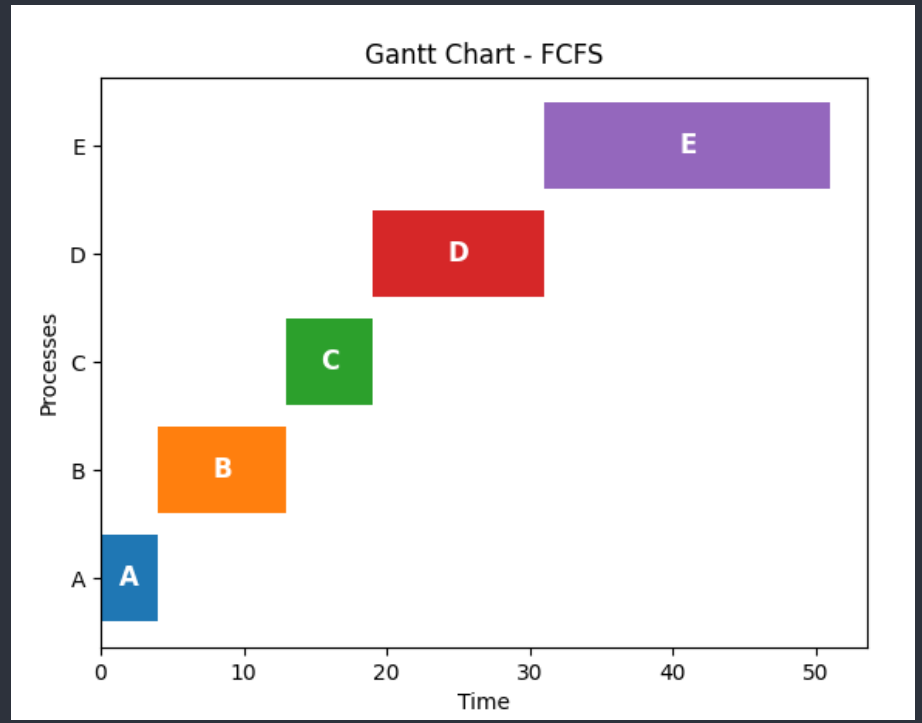
</ FCFS(先到先行)模擬

先對程序A~E進行到達時間排序，並計算各自的等待時間，並在行程排程完之後加總等待時間後，除以行程量導出平均等待時間。

```
# Ensure processes are sorted by arrival time before execution
self.processes.sort(key=lambda p: p.arrival_time)
```

```
def display_avg_waiting_time(self, title):
    self.processes.sort(key=lambda p: p.pid)  # Ensure ordering before computing
    avg_waiting_time = sum(p.waiting_time for p in self.processes) / len(self.processes)
    print(f"{title} - Average Waiting Time: {avg_waiting_time:.2f}")
```

```
FCFS - Average Waiting Time: 7.00
```



1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ RR (Q=3) 模擬

```
# Add newly arrived processes to queue
while index < n and self.processes[index].arrival_time <= self.time_counter:
    queue.append(self.processes[index])
    index += 1
13 minutes ago • Uncommitted changes
if queue:
    process = queue.popleft()
    execution_time = min(quantum, process.remaining_time)

    # Schedule process execution
    self.schedule.append((process.pid, self.time_counter, self.time_counter + execution_time))
    self.time_counter += execution_time
    process.remaining_time -= execution_time

    # Add newly arrived processes after execution
    while index < n and self.processes[index].arrival_time <= self.time_counter:
        queue.append(self.processes[index])
        index += 1

    # If process is not finished, put it back in the queue
    if process.remaining_time > 0:
        queue.append(process)
    else:
        process.completion_time = self.time_counter
        process.turnaround_time = process.completion_time - process.arrival_time
        process.waiting_time = process.turnaround_time - process.burst_time
        completed += 1
else:
    # If no process is ready, jump to the next process's arrival time
    self.time_counter = self.processes[index].arrival_time
```

```
# Ensure processes are sorted by arrival time before execution
self.processes.sort(key=lambda p: p.arrival_time)
```

先對程序A~E進行到達時間排序，並計算各自的等待時間，並用雙向佇列排程，以三個系統時間為單位執行，並照到達時間為順序，依序執行三個系統時間，若提前運行完，則由後續程序補上。

平均運行時間由周轉時間減去執行時間，周轉時間則由每個程序執行完後的時間減去到達時間。

1 0 1 1　0 1 1　0 1　1 0 1 1 0 0 1　1 0　1 1 0 1 1　0 1 1　0 1　1 1 0 1 1 0　1 1 0 1 1 1　1 1 0 1

</ SJF(最短行程優先)模擬

先對程序A~E進行到達時間、執行時間排序。

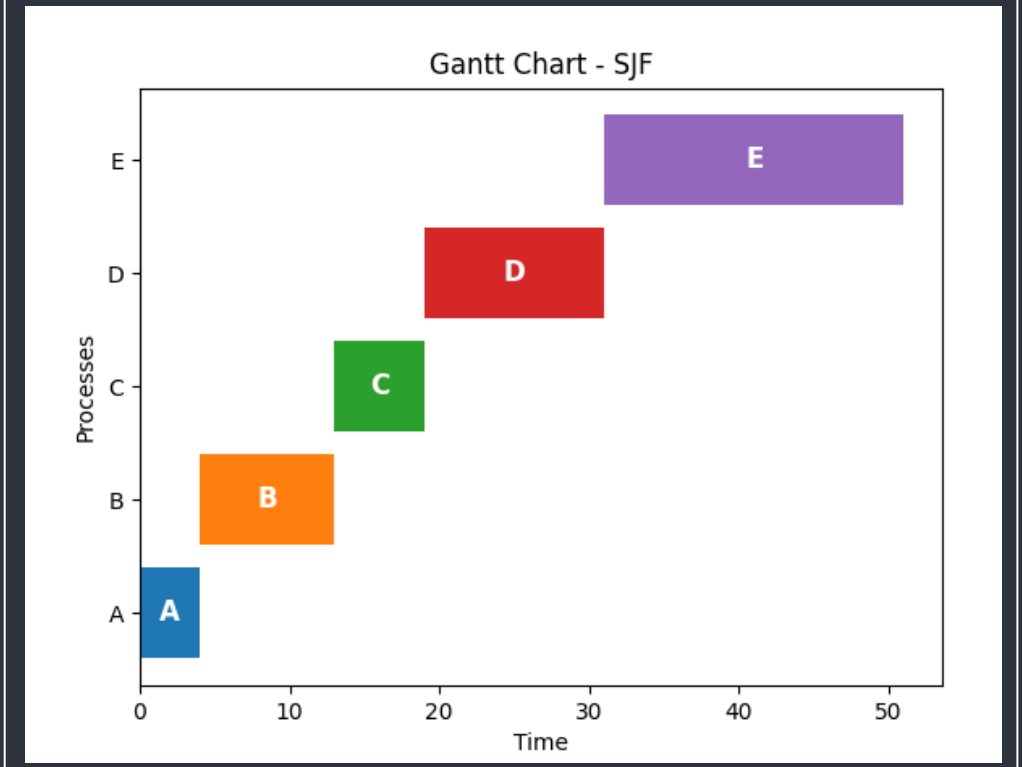
```
# Ensure processes are sorted by arrival time & burst time first
self.processes.sort(key=lambda p: (p.arrival_time, p.burst_time))
```

各行程的等待時間則由當下的系統總時間減去到達時間，之後再加總等待時間除以行程總量。

```
process.waiting_time = self.time_counter - process.arrival_time
```

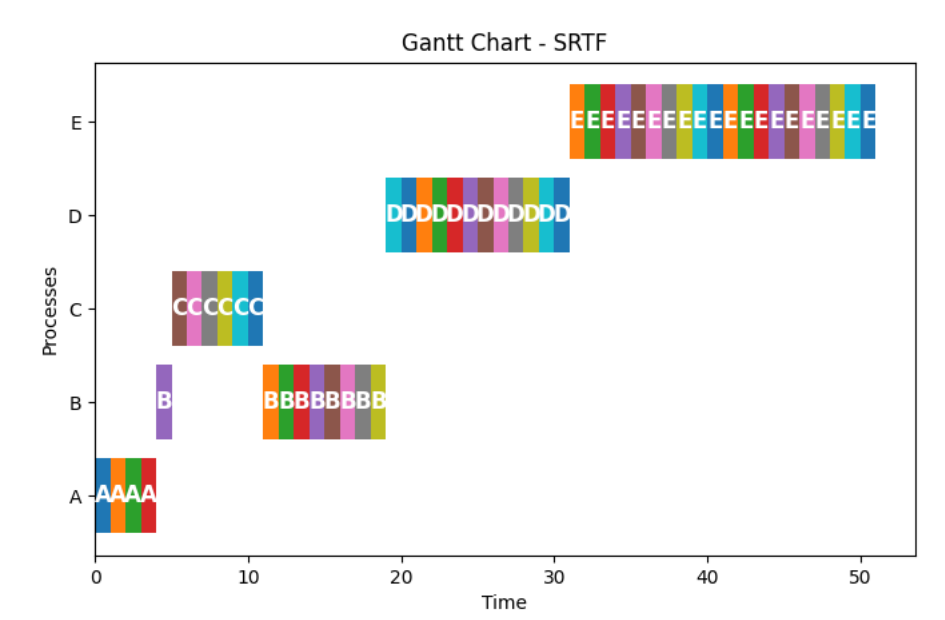
```
# Calculate and display average waiting time
avg_waiting_time = total_waiting_time / n
```

```
SJF - Average Waiting Time: 7.00
```



</ SRTF(最短剩餘先行) 模擬

```
SRTF - Average Waiting Time: 6.60
```




```python
# Ensure processes are sorted by arrival time before execution
self.processes.sort(key=lambda p: p.arrival_time)
```

先對程序A~E進行到達時間排序，並依據系統時間將到達形成陸續加入排程。

```python
completed = 0
remaining_processes = self.processes[:]

while completed < len(self.processes):
    # 將已到達的行程加入 ready_queue
    while remaining_processes and remaining_processes[0].arrival_time <= self.time_counter:
        self.ready_queue.append(remaining_processes.pop(0))

    # 選擇剩餘時間最短的行程執行
    if self.ready_queue:
        self.ready_queue.sort(key=lambda p: p.remaining_time)
        process = self.ready_queue[0]  # 選擇當前剩餘時間最短的行程
        process.remaining_time -= 1
        self.schedule.append((process.pid, self.time_counter, self.time_counter + 1))
        self.time_counter += 1
```

程序會分割為一個個系統時執行，並優先執行剩餘執行時間較少的程序。
若程式運行完畢，則等待時間為:
完成時間-到達時間-執行時間

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ 排程方法比較

排程演算法	優點	缺點
SCSF (FCFS, First-Come, First-Served)	◆ 簡單易實作。 ◆ 適合長時間運行的批次處理系統。 ◆ 沒有饑餓 (Starvation) 問題。	◆ 短行程可能要等很久 (導致高等待時間)。 ◆ 平均等待時間可能很長，影響系統響應。 ◆ 容易產生 「**Convoy Effect**」 (短行程受長行程影響延遲)。
RR (Round-Robin, q=3)	◆ 適合 **交互系統 (Interactive Systems)**，確保所有行程獲得公平執行機會。 ◆ 沒有 **飢餓 (Starvation)** 問題。 ◆ 平均響應時間較短，適合多使用者環境。	◆ 若時間片過小，則 **上下文切換 (Context Switch) 開銷大**，影響效能。 ◆ 若時間片過大，則可能變成接近 FCFS。 ◆ 若有長行程，它仍然會等待多個時間片，影響系統效率。
SJF (Shortest Job First)	◆ 平均等待時間最低，適合批次作業處理。 ◆ 效率高，能夠提升吞吐量 (Throughput)。 ◆ 適合短行程密集的環境，執行效率較高。	◆ **飢餓問題 (Starvation)**：長行程可能長時間不被執行。 ◆ 需要知道確切的執行時間 (Burst Time)，難以應用於即時系統。 ◆ 動態變動的環境難以應用，行程長度估算不準會影響效果。
SRTF (Shortest Remaining Time First)	◆ 比 SJF 更適合動態環境，因為會根據 **剩餘執行時間** 動態調整。 ◆ 最小化平均等待時間和周轉時間。 ◆ 提供更高的系統效率。	◆ **可能會導致飢餓 (Starvation)**，因為短行程可能不斷插隊，導致長行程一直延遲。 ◆ **上下文切換 (Context Switch) 次數多**，可能影響系統效能。 ◆ 需要準確的估算剩餘執行時間，這可能不容易實作。

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1