VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY

HO CHI MINH CITY UNIVERSITY OF SCIENCE

INFORMATION TECHNOLOGY DEPARTMENT

Assignment 02.03

Topic: Write a program for running the CPU Scheduling: FCFS, RR, SJF, Priority

---

Course: Hệ điều hành

Student:
Đinh Nguyễn Gia Bảo (22127027)
Nguyễn Công Tuấn (22127436)

Lecturer:
Thái Hùng Văn

21st April 2024

Contents

1 Team Information

STUDENT INFORMATION TABLE

Student	ID
Đinh Nguyễn Gia Bảo	22127027
Nguyễn Công Tuấn	22127436

TASK TABLE

No.	Task	Person in charge	Completion
1	**FCFS** (First Come First Serve)	Nguyễn Công Tuấn	100%
2	**RR** (Round Robin)	Đinh Nguyễn Gia Bảo	100%
3	**SJF** (Shortest Job First)	Đinh Nguyễn Gia Bảo	100%
4	**Priority**	Nguyễn Công Tuấn	100%
5	**SRTN** (Shortest Remaining Time Next) (**Bonus**)	Nguyễn Công Tuấn	100%
6	Write the report	Cả hai	100%

2 Requirement

Write a program that takes a text file Input.txt containing information about the processes that need to be coordinated, determines the CPU scheduling result, and the corresponding parameters according to the presented strategies (FCFS, RR, SJF, Priority), with the result for each strategy saved in a corresponding text file.

Note: Each line of the Input file contains 3 or 4 pieces of information: <Process Name> <Arrival Time> <Processing Time> [<Priority>]; only the first line contains 2 pieces of information: the number of processes and the time quantum of the Round Robin strategy.

For example, with 03 processes P1, P2, P3, the information might be as follows:

Process	Arrival Time	CPU Burst	Priority	Quantum=4
P1	0	24	3	Thời gian xử lý của modul điều phối rất nhỏ và xem như là 0 (tức không xét), các tiến trình này cũng không rơi vào trạng thái Blocked
P2	1	5	2	
P3	2	3	1	

Then the content of the Input.txt file would be:

```
3    4
P1   0    24   3
P2   1    5    2
P3   2    3    1
```

In this lab, we use:

- Programming Language: **Python**
- Code editor: **Visual Studio Code**
- CPU Scheduling Algorithms:
  - **FCFS** (First Come First Serve)
  - **RR** (Round Robin)
  - **SJF** (Shortest Job First)
  - **Priority**
  - **SRTN** (Shortest Remaining Time Next) (**Bonus**)

3 CPU Scheduling

3.1 FCFS (First Come First Serve)

- The First Come First Serve (FCFS) scheduling algorithm is one of the simplest CPU scheduling algorithms.
- In FCFS, processes are executed in the order they arrive, with the first process to arrive being the first to be executed.
- This algorithm is non-preemptive, meaning once a process starts executing, it continues until it completes its CPU burst and it is easy to implement but may result in poor average waiting times, especially for processes with long CPU burst times.

3.1.1 Step-by-Step implementation

1. Arrange the processes in the order they arrive.
2. Set the initial start time and end time of the process to 0.
3. For each process:
   - Calculate the start time of the process by taking the maximum of the previous process's end time and the current process's arrival time.
   - Calculate the end time of the process by adding the CPU burst time to its start time.
4. Store the scheduling information, including the process's name, start time, and end time.
5. Set the completion time of each process to its end time.
6. Determine the waiting time for each process.
7. Calculate the turnaround time for each process.
8. Save the scheduling information, turnaround time, and waiting time to an output file.

3.1.2 Code example

```
def FCFS():
    processes_group.sort(key=lambda x: x.arrival_time) # sort arrival_time for all processes

    scheduling = []

    start_time_of_process = 0
    end_time_of_process = 0

    for i in range(num_processes):
        start_time_of_process = max(end_time_of_process, processes_group[i].arrival_time)
```

```
11          end_time_of_process = start_time_of_process + processes_group[i].
    cpu_burst
12
13          scheduling.append((start_time_of_process, processes_group[i].name,
    end_time_of_process))
14
15          processes_group[i].set_completion_time(end_time_of_process)
16
17      waiting_time = calculate_waiting_time([process.completion_time for process
    in processes_group])
18      turnaround_time = calculate_turnaround_time([process.completion_time for
    process in processes_group])
19
20      write_output("FCFS.txt", scheduling, turnaround_time, waiting_time)
```

The code segment above shows that:

- Sort Processes by Arrival Time.
- Iterate Through Processes:
  - Determine each process's start and end time based on arrival time and CPU burst.
  - Record scheduling information.
  - Calculate completion time.
- Calculate Waiting Time and Turnaround Time.
- Write Output to File.

3.2 RR (Round Robin)

- Round Robin (RR) is a preemptive algorithm designed for time-sharing systems.
- In RR, each process is assigned a fixed time quantum, and the CPU scheduler switches between processes once their quantum expires, regardless of whether the process has finished its execution or not.
- This algorithm ensures fairness by giving each process an equal opportunity to execute and prevents any single process from monopolizing the CPU for an extended period. However, RR may lead to increased context-switching overhead, especially with smaller time quantum values.

3.2.1 Step-by-Step implementation

1. Arrange the processes in the order they arrive.
2. Initialize an empty list to store the scheduling information.
3. Create a copy of the list of processes to keep track of the remaining processes.
4. Set the initial start and end quantum times to 0.

5. While there are remaining processes:
    - For each process in the list of remaining processes:
        - If the process has arrived by the end of the current quantum:
            * If the process has remaining time greater than the quantum:
                · Update the start and end quantum times for the process.
                · Append the scheduling information for the process to the list.
                · Decrease the remaining time of the process by the quantum.
            * Else:
                · Update the start and end quantum times for the process.
                · Append the scheduling information for the process to the list.
                · Update the remaining time of the process to 0.
                · Set the completion time for the process.
                · Increment the count of completed processes.
6. Remove processes with remaining time 0 from the list of remaining processes.
7. If no process was scheduled during the current quantum and there are remaining processes:
    - Increment the end quantum time by 1 to handle idle time.
8. Calculate the waiting time and turnaround time for each process.
9. Save the scheduling information, turnaround time, and waiting time to an output file.
10. Reset the remaining time of all processes to their original values.

3.2.2 Code example

```
def RR():
    processes_group.sort(key=lambda x: x.arrival_time) # sort arrival_time for all processes

    scheduling = []
    remaining_processes = processes_group[:]

    start_quantum_time_of_process = 0
    end_quantum_time_of_process = 0

    process_done = 0

    is_any_process_did = False

    while remaining_processes:
        for process in range(len(remaining_processes)):
            if remaining_processes[process].arrival_time <= end_quantum_time_of_process:
                if remaining_processes[process].remaining_time > quantum_time:
                    is_any_process_did = True

```

```
                    start_quantum_time_of_process = end_quantum_time_of_process
                    end_quantum_time_of_process += quantum_time

                    scheduling.append((start_quantum_time_of_process,
remaining_processes[process].name, end_quantum_time_of_process))

                    remaining_processes[process].remaining_time -= quantum_time

                else:
                    is_any_process_did = True
                    start_quantum_time_of_process = end_quantum_time_of_process
                    end_quantum_time_of_process += remaining_processes[process].
remaining_time

                    scheduling.append((start_quantum_time_of_process,
remaining_processes[process].name, end_quantum_time_of_process))

                    remaining_processes[process].remaining_time = 0
                    remaining_processes[process].set_completion_time(
end_quantum_time_of_process)

                    process_done += 1

        remaining_processes = [process for process in remaining_processes if
process.remaining_time > 0]

        if process_done < num_processes and is_any_process_did == False:
            start_quantum_time_of_process = end_quantum_time_of_process
            end_quantum_time_of_process += 1

        is_any_process_did = False

    waiting_time = calculate_waiting_time([process.completion_time for process
in processes_group])
    turnaround_time = calculate_turnaround_time([process.completion_time for
process in processes_group])

    write_output("RR.txt", scheduling, turnaround_time, waiting_time)

    for i in range(num_processes):
        processes_group[i].set_remaining_time_again()
```

The code segment above shows that:

- Sort Processes by Arrival Time.
- Initialize Variables.
- Iterate through processes:
  - Execute each process based on the quantum time.
  - Update scheduling and remaining time.
- Calculate the Waiting Time and Turnaround Time.

- Write Output.
- Reset Remaining Time.

3.3 SJF (Shortest Job First)

- Shortest Job First (SJF) is a preemptive algorithm that selects the process with the smallest burst time or execution time for the next execution.
- In SJF, the process that requires the least amount of CPU time is given priority, allowing for efficient utilization of CPU resources and minimizing the average waiting time and turnaround time for all processes. If a new process arrives with a shorter burst time than the currently executing process, it can preempt the CPU and start executing immediately, resulting in optimal scheduling and throughput.
- SJF preemptive scheduling is commonly used in interactive and time-sharing systems where responsiveness and efficiency are critical.

3.3.1 Step-by-Step implementation

1. Arrange the processes in the order they arrive.
2. Create an empty list to store the scheduling information.
3. Create a copy of the list of processes to keep track of the remaining processes.
4. Set the current time to 0.
5. While there are remaining processes:
    - Check for processes that have arrived by the current time.
    - If there are arriving processes:
        - Select the process with the shortest CPU burst time among the arriving processes.
        - Record the start and end times of the selected process's execution.
        - Update the current time to the end of the selected process's execution.
        - Remove the selected process from the list of remaining processes.
    - If no processes have arrived:
        - Increment the current time by 1 unit.
6. Calculate the waiting time and turnaround time for each process based on their completion times.
7. Save the scheduling information, turnaround time, and waiting time to an output file.
8. Reset the remaining time of all processes to their original values.

3.3.2 Code example

```
def SJF():
    processes_group.sort(key=lambda x: x.arrival_time) # Sort processes by arrival time

    scheduling = []
    remaining_processes = processes_group[:]
    current_time = 0

    while remaining_processes:
        arriving_processes = [process for process in remaining_processes if process.arrival_time <= current_time]

        if arriving_processes:
            process_min_cpu_burst = min(arriving_processes, key=lambda process: process.cpu_burst)

            start_of_process = current_time
            end_of_process = current_time + process_min_cpu_burst.cpu_burst

            scheduling.append((start_of_process, process_min_cpu_burst.name, end_of_process))

            current_time = end_of_process

            remaining_processes.remove(process_min_cpu_burst)
        else:
            current_time += 1

    waiting_time = calculate_waiting_time([process.completion_time for process in processes_group])
    turnaround_time = calculate_turnaround_time([process.completion_time for process in processes_group])

    write_output("SJF.txt", scheduling, turnaround_time, waiting_time)

    for process in processes_group:
        process.set_remaining_time_again()
```

The code segment above shows that:

- Sort Processes by Arrival Time.
- While there are remaining processes:
  - Identify arriving processes.
  - Select the process with the shortest CPU burst time.
  - Record scheduling information and update time.
  - Remove the executed process.
- Calculate the Waiting Time and Turnaround Time.
- Write Output.

- Reset Remaining Time.

3.4 Priority

- Priority is an algorithm where each process is assigned a priority. The CPU scheduler selects the process with the highest priority for execution.
- In this scheduling algorithm, the priority of a process can be determined based on factors such as the importance of the task, its deadline, or its resource requirements. Processes with higher priority are given precedence over those with lower priority, ensuring that critical tasks are completed promptly.
- Priority scheduling can be either preemptive or non-preemptive, depending on whether the currently executing process can be interrupted by a higher priority process.

3.4.1 Step-by-Step implementation

1. Arrange the processes in the order they arrive.
2. Create an empty list to store the scheduling information.
3. Initialize an empty list to store processes that have arrived but not yet been scheduled.
4. Set the current time and start time to 0.
5. Initialize the number of loaded processes and the current process ID.
6. While there are processes to load and the arrival time of the next process matches the current time, add the process to the list of remaining processes and update the process count.
7. While there are remaining processes or processes loaded:
    - If there are remaining processes:
        - Select the process with the highest priority from the list of remaining processes.
        - If the current process is different from the selected process:
            - If the start time is different from the current time and the current process is in the remaining processes list, record the scheduling information.
            - Update the start time and current process ID.
        - Decrement the remaining time of the selected process and advance the current time.
        - If there are no remaining processes, update the current process to the next loaded process.
        - If the remaining time of the selected process is zero, record the scheduling information and remove the process from the list of remaining processes.
8. Calculate the waiting time and turnaround time for each process based on their completion times.
9. Save the scheduling information, turnaround time, and waiting time to an output file.

3.4.2 Code example

```
def Priority():
    processes_group.sort(key=lambda x: x.arrival_time)  # Sort processes by arrival time

    scheduling = []
    remaining_processes = []

    current_time = 0
    start_time = 0

    process_loaded_number = 0
    id_current_process = processes_group[process_loaded_number]

    while process_loaded_number < num_processes or remaining_processes:
        while process_loaded_number < num_processes and processes_group[process_loaded_number].arrival_time == current_time:
            remaining_processes.append(processes_group[process_loaded_number])
            process_loaded_number += 1

        if remaining_processes:
            process_highest_priority = min(remaining_processes, key=lambda process: process.priority)

            if id_current_process != process_highest_priority:
                if start_time != current_time and id_current_process in remaining_processes:
                    scheduling.append((start_time, id_current_process.name, current_time))
                start_time = current_time
                id_current_process = process_highest_priority

            process_highest_priority.remaining_time -= 1
            current_time += 1

            if not remaining_processes:
                id_current_process = processes_group[process_loaded_number]

            if process_highest_priority.remaining_time == 0:
                scheduling.append((start_time, id_current_process.name, current_time))
                start_time = current_time
                remaining_processes.remove(process_highest_priority)

        else:
            current_time += 1

    waiting_time = calculate_waiting_time([process.completion_time for process in processes_group])
    turnaround_time = calculate_turnaround_time([process.completion_time for process in processes_group])

    write_output("Priority.txt", scheduling, turnaround_time, waiting_time)
```

The code segment above shows that:

- Sort Processes by Arrival Time.
- While there are remaining processes or loaded processes:
  - Load arriving processes.
  - Select the process with the highest priority.
  - Record scheduling information and update time.
  - Remove the executed process if completed.
- Calculate the Waiting Time and Turnaround Time.
- Write Output.

3.5 SRTN (Shortest Remaining Job First)

- Shortest Remaining Time Next (SRTN) is a preemptive variant of the Shortest Job First (SJF) algorithm.
- In SRTN, the process with the smallest remaining burst time is selected for execution next. If a new process arrives with a shorter burst time than the currently executing process, it preempts the CPU and starts executing immediately.
- This scheduling algorithm aims to minimize the remaining time for each process, leading to efficient resource utilization and reduced waiting times. SRTN is particularly effective for interactive and time-sharing systems where responsiveness is crucial and short tasks need to be prioritized for execution.

3.5.1 Step-by-Step implementation

1. Arrange the processes in the order they arrive.
2. Create an empty list to store the scheduling information.
3. Initialize an empty list to store processes that have arrived but not yet been scheduled.
4. Set the current time and start time to 0.
5. Initialize the number of loaded processes and the current process ID.
6. While there are processes to load and the arrival time of the next process matches the current time, add the process to the list of remaining processes and update the process count.
7. While there are remaining processes or processes loaded:
   - If there are remaining processes:
     - Select the process with the shortest remaining time from the list of remaining processes.
     - If the current process is different from the selected process:

    - ∗ If the start time is different from the current time and the current process is in the remaining processes list, record the scheduling information.
    - ∗ Update the start time and current process ID.
  - Decrement the remaining time of the selected process and advance the current time.
  - If there are no remaining processes, update the current process to the next loaded process.
  - If the remaining time of the selected process is zero, record the scheduling information, set the completion time for the process, and remove it from the list of remaining processes.

8. Calculate the waiting time and turnaround time for each process based on their completion times.

9. Save the scheduling information, turnaround time, and waiting time to an output file.

10. Reset the remaining time of all processes to their original values.

3.5.2 Code example

```
def SRTN():
    processes_group.sort(key=lambda x: x.arrival_time)  # Sort processes by arrival time

    scheduling = []
    remaining_processes = []

    current_time = 0
    start_time = 0

    process_loaded_number = 0
    id_current_process = processes_group[process_loaded_number]

    while process_loaded_number < num_processes or remaining_processes:
        while process_loaded_number < num_processes and processes_group[process_loaded_number].arrival_time == current_time:
            remaining_processes.append(processes_group[process_loaded_number])
            process_loaded_number += 1

        if remaining_processes:
            process_min_remaining_time = min(remaining_processes, key=lambda process: process.remaining_time)

            if id_current_process != process_min_remaining_time:
                if start_time != current_time and id_current_process in remaining_processes:
                    scheduling.append((start_time, id_current_process.name, current_time))
                start_time = current_time
                id_current_process = process_min_remaining_time

            process_min_remaining_time.remaining_time -= 1
            current_time += 1
```

```
29
30          if not remaining_processes:
31              id_current_process = processes_group[process_loaded_number]
32
33          if process_min_remaining_time.remaining_time == 0:
34              scheduling.append((start_time, id_current_process.name,
    current_time))
35              id_current_process.set_completion_time(current_time)
36              start_time = current_time
37              remaining_processes.remove(process_min_remaining_time)
38
39      else:
40          current_time += 1
41
42
43   waiting_time = calculate_waiting_time([process.completion_time for process
    in processes_group])
44   turnaround_time = calculate_turnaround_time([process.completion_time for
    process in processes_group])
45
46   write_output("SRTN.txt", scheduling, turnaround_time, waiting_time)
47
48   for process in processes_group:
49       process.set_remaining_time_again()
```

- Sort Processes by Arrival Time.
- While there are remaining processes or loaded processes:
  - Load arriving processes.
  - Select the process with the shortest remaining time.
  - Record scheduling information and update time.
  - Remove the executed process if completed.
- Calculate the Waiting Time and Turnaround Time.
- Write Output.

4 Reference

[1] CPU Scheduling in Operating Systems.
[2] CPU Scheduling - HCMUS OS
[3] CPU Scheduling - University of Illinois Chicago