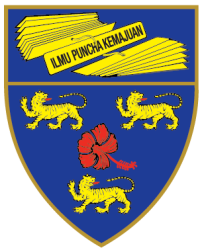


UNIVERSITI MALAYA

WIA2004 Operating Systems

Lab 2: SJF CPU Scheduling Algorithms

Group : F5U

Occ: 10

Lecturer: Dr. Fazidah Othman

GROUP MEMBERS	MATRICS NUMBER
ZHANG ZHIYANG	S2193685
MEILIN LI	S2174975
HUMYRA TASMIA	S2176677
HUSNA NADIAH BINTI ROSTHAM	22060027
HUSNA BINTI IHSANUDDIN	U2102305

Table of Content

1.0 Introduction

CPU scheduling algorithms play a crucial role in managing process execution within computer systems. Among these algorithms, Shortest Job First (SJF), also known as Shortest Job Next (SJN), is a non-preemptive scheduling algorithm. SJF prioritizes processes based on their CPU cycle time, making it particularly suitable for batch environments where the estimated CPU time required is known at the start of a job.

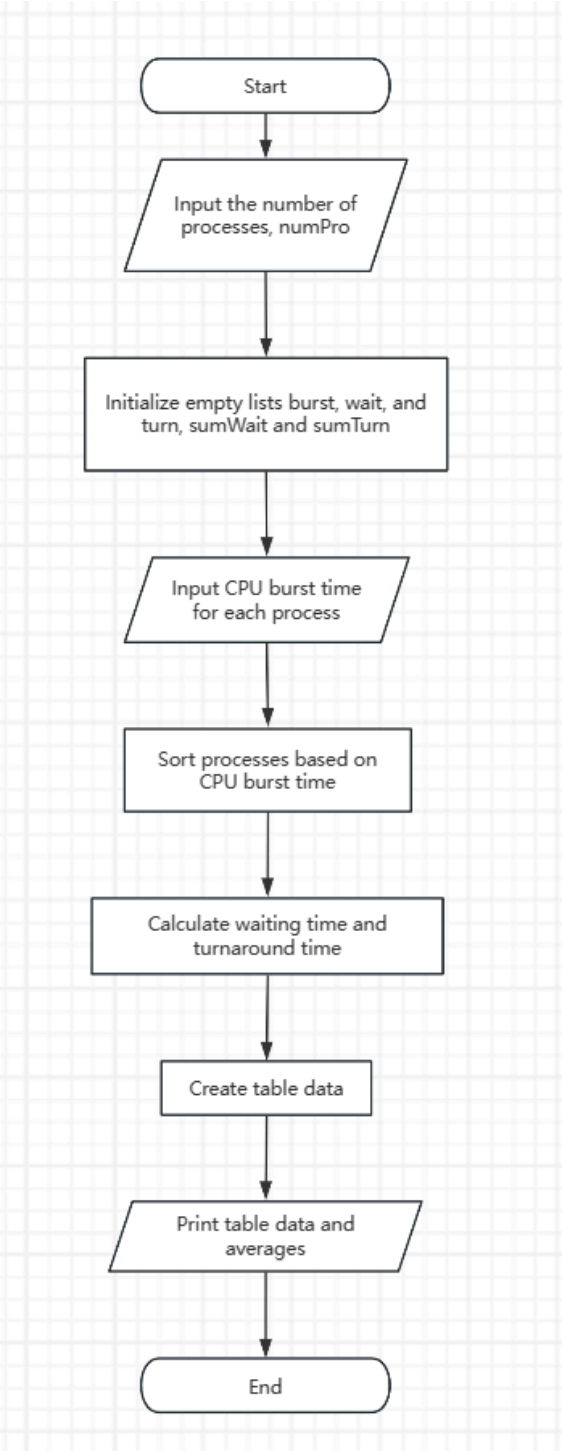
SJF handles based on the length of their CPU cycle time, making it easier to implement in batch environments where the estimated CPU time required is stated by the user at the start of a job.

Here we are assuming that the arrival time for all processes is the same. The turnaround time for each process is defined as the total time it takes from submission to the system until completion, calculated as the finish time of each job minus its arrival time. Turnaround time = each job's finish time - arrival time.

The program simulates the SJF CPU scheduling algorithm by accepting inputs such as the number of processes and their corresponding CPU burst times. It then calculates the waiting time and turnaround time for each process based on their arrival order, providing insights into the performance of the SJF algorithm.

2.0 Methodology

Flowchart



Pseudocode

1. Import the tabulate library.

2. Define the main function:
   2.1. Prompt the user to enter the number of processes (numPro).
   2.2. Initialize an empty list named burst to store burst times of processes.
   2.3. Initialize a list named wait with numPro elements, all initialized to 0.
   2.4. Initialize a list named turn with numPro elements, all initialized to 0.
   2.5. Initialize a variable named sumWait and set it to 0.
   2.6. Initialize a variable named sumTurn and set it to 0.

   2.7. Iterate numPro times:
      2.7.1. Prompt the user to enter the burst time for process i and append it to the burst list.

   2.8. Create a list named process containing the process IDs [0, 1, 2, ..., numPro - 1].

   2.9. Sort the processes based on burst time:
      2.9.1. Iterate over the processes from 0 to numPro - 2:
         2.9.1.1. Iterate over the processes from 0 to numPro - i - 2:
            2.9.1.1.1. If burst[j] > burst[j + 1], swap the burst times and process IDs.

   2.10. Calculate the waiting time for each process:
      2.10.1. Iterate over the processes starting from the second process:
         2.10.1.1. Set wait[i] to the sum of burst times of all previous processes.
         2.10.1.2. Add wait[i] to sumWait.

   2.11. Calculate the turnaround time for each process:
      2.11.1. Iterate over all processes:
         2.11.1.1. Set turn[i] to burst[i] + wait[i].
         2.11.1.2. Add turn[i] to sumTurn.

   2.12. Create a table_data list to store process information.
   2.13. Iterate over each process:
      2.13.1. Append a list containing process ID, burst time, waiting time, and turnaround time to table_data.

   2.14. Print the table using tabulate, along with average waiting time and average turnaround time.

3. If the script is run directly (not imported), call the main function.

3.0 Implementation

3.1 Coding

```
from tabulate import tabulate

def main():
    numPro = int(input("Enter number of processes: "))
    burst = []
    wait = [0] * numPro
    turn = [0] * numPro
    sumWait = 0
    sumTurn = 0

    for i in range(numPro):
        burst_time = int(input(f"Enter Burst time for P[{i + 1}]: "))
        burst.append(burst_time)

    # Sorting processes according to burst time
    process = list(range(numPro))
    for i in range(numPro - 1):
        for j in range(numPro - i - 1):
            # swap the burst time
            if burst[j] > burst[j + 1]:
                burst[j], burst[j + 1] = burst[j + 1], burst[j]

                # swap the process order
                process[j], process[j + 1] = process[j + 1], process[j]

    # Calculate waiting time
    for i in range(1, numPro):
        wait[i] = wait[i - 1] + burst[i - 1]
        sumWait += wait[i]
```

```
    # Calculate turnaround time
    for i in range(numPro):
        turn[i] = burst[i] + wait[i]
        sumTurn += turn[i]

    table_data = []
    for i in range(numPro):
        table_data.append(["P[{}]".format(process[i] + 1), burst[i], wait[i], turn[i]])

    headers = ["Process ID", "Burst Time", "Waiting time", "Turnaround Time"]
    print(tabulate(table_data, headers=headers, tablefmt="grid"))

    print("Average Waiting Time: {:.2f}".format(sumWait / numPro))
    print("Average Turnaround Time: {:.2f}".format(sumTurn / numPro))


if __name__ == "__main__":
    main()
```

3.2 Sample output

```
+--------------+--------------+----------------+--------------------+
| Process ID   |   Burst Time |   Waiting time |   Turnaround Time |
+==============+==============+================+====================+
| P[4]         |            3 |              0 |                  3 |
+--------------+--------------+----------------+--------------------+
| P[1]         |            6 |              3 |                  9 |
+--------------+--------------+----------------+--------------------+
| P[3]         |            7 |              9 |                 16 |
+--------------+--------------+----------------+--------------------+
| P[2]         |            8 |             16 |                 24 |
+--------------+--------------+----------------+--------------------+
Average Waiting Time: 7.00
Average Turnaround Time: 13.00
```

4.0 Conclusion

In conclusion, we found that the Shortest Job First (SJF) scheduling algorithm effectively organizes tasks by their length, which can lead to shorter waiting times and better overall system performance. We saw firsthand how SJF can make things run more smoothly and quickly, thanks to detailed calculations and visual aids. However, we also noticed a downside: longer tasks might get stuck waiting if shorter ones keep coming in, a problem known as "starvation." This contrasts with the First Come First Serve (FCFS) method, which treats all tasks equally but might not be as efficient. To sum up, while SJF can greatly improve efficiency by reducing wait and turnaround times, it's not perfect because it might unfairly delay longer tasks. This exercise shows it's crucial to pick the right scheduling method for your system's needs.

5.0 References

- April 2020 IJARCCE 9(4):41-45 Analysis of Preemptive Shortest Job First (SJF) Algorithm in CPU Scheduling.10.17148/IJARCCE.2020.9408
- IJCSMC, Vol. 8, Issue. 5, May 2019, pg.176 – 181 COMPARATIVE ANALYSIS BETWEEN FIRST-COME-FIRST-SERVE (FCFS) AND SHORTEST-JOB-FIRST (SJF) SCHEDULING ALGORITHMS https://d1wqtxts1xzle7.cloudfront.net/59419541/V8I520193520190528-105020-ffu0gq-libre.pdf?1559037820=&response-content-disposition=inline%3B+filename%3DCOMPARATIVE_ANALYSIS_BETWEEN_FIRST_COME.pdf&Expires=1712116570&Signature=DeY0hpd6b5YA7e7uCtt5neL7uVvBkOIv8ZNPOg7~clvJyY8Fr6s6yd-z-Js5P~1~ER7Da4ovy1KiYITJJOo~reF6mY5CPDHpqpkbYdNBnwQC7jhGzC9~jqtRn2FhP9DxfL8JYII5Y1dotRzgX6HZK5be2I6Bq1pEJHJkJlq4sFN7tFoi6hYak5jw4bIZP73KaZ7C37KzvKwyLzgphWkI4kN~xKucZ9CmRD3GwqEXXYrm-JEmIYKANStg3pKWo61Qm6pyxG~FZdWp2f40BC8423x~L4t9v7svaCLwFJGvj1BF0ph-79wxhiHSByH~NpHagptOrVqMkyAXQAfe8sx0VQ__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA