6/13/2025

FINAL PROJECT

OPERATING SYSTEM LAB

GROUP # 5

FAHAD QAAYYUM

SAIF ALI

ZEESHAN HAIDER

CONTENTS

Introduction

1.1 Project Overview

This project, titled **"SJF and Round Robin Comparative Analyzer"**, is a Python-based simulation of key Operating System (OS) concepts. It focuses primarily on comparing two fundamental CPU scheduling algorithms — **Shortest Job First (SJF)** and **Round Robin (RR)** — using various performance metrics. In addition to scheduling, the project implements **Memory Management techniques**, **Deadlock Detection**.

The simulation provides a practical insight into how operating systems manage processes and resources. Through this project, core concepts are demonstrated using simple yet effective Python scripts and graphical visualizations using Matplotlib

1.2 Objectives

- To understand and simulate **CPU Scheduling Algorithms**, specifically SJF and Round Robin.
- To compare these algorithms based on:
  - Average Waiting Time
  - Turnaround Time
  - Context Switches
  - CPU Utilization
  - Throughput
- To simulate **Memory Allocation Techniques** such as **First Fit** and **Best Fit**.
- To implement a basic **Deadlock Detection** mechanism using safe sequence logic.
- To demonstrate **Inter-Process Communication (IPC)** via **Shared Memory** using Python's multiprocessing module.
- To visualize performance metrics through graphs for better interpretation.

1.3 Technologies Used

Technology	Purpose
Python (v3.10+)	Core logic, simulation, scheduling, IPC
Matplotlib	Data visualization and graph plotting
Multiprocessing	Shared memory & IPC simulation
Jupyter Notebook / VS Code	Development environment

Technology	Purpose
Terminal / CLI	Running the simulations

Scheduling Algorithms

The implementation and evaluation of two fundamental CPU scheduling algorithms — **Shortest Job First (SJF)** and **Round Robin (RR)**. These are analyzed using real-time simulation, and their performance is compared using multiple metrics.

2.1 Shortest Job First (SJF)

SJF is a non-preemptive scheduling algorithm that selects the process with the shortest burst time among the available ones. It reduces average waiting time but may lead to starvation of longer processes.

How It Works

- All processes are sorted by their arrival time.
- Among all arrived processes, the one with the **smallest burst time** is selected.
- The process runs to completion.
- Repeat until all processes are executed.

Code Snippet

```
Python
```

```
def sjf_scheduling(processes):
    ...
    # Select process with minimum burst that has arrived
    ...
```

Advantages

- Minimal average waiting time (for short jobs)
- Simple to implement

Disadvantages

- Non-preemptive, so may cause starvation
- Not ideal for real-time systems

2.2 Round Robin (RR)

Round Robin is a preemptive scheduling algorithm that gives each process a fixed time slice (quantum). If the process doesn't finish, it's sent to the back of the queue.

How It Works

- All processes are added to a queue based on arrival time.
- Each process is allowed to run for a fixed **quantum**.
- If it's not finished, it's placed at the end of the queue.
- This continues until all processes are completed.

Code Snippet

Python

```
def round_robin_scheduling(processes, quantum=2):
    ...
    # Run process for min(quantum, remaining_time)
    ...
```

Advantages

- Fair allocation of CPU
- Suitable for time-sharing systems

Disadvantages

- Larger turnaround time than SJF
- Performance depends heavily on chosen quantum

2.3 Performance Metrics

Metric	Description
Average Waiting Time	Total waiting time / number of processes
Average Turnaround Time	Total turnaround time / number of processes
Context Switches	Number of times CPU switches from one process to another
CPU Utilization	Ratio of active CPU time to total time
Throughput	Number of processes completed per unit time

Memory Management

The memory allocation strategies implemented in the project: **First Fit** and **Best Fit**. Both strategies are used to allocate memory blocks to processes, and the allocation is visualized using bar graphs for better understanding.

3.1 Introduction to Memory Management

Memory management is a critical function of the operating system, which handles the allocation and deallocation of memory to various processes. Efficient memory allocation reduces internal fragmentation and improves system performance.

3.2 First Fit Allocation

First Fit is a straightforward memory allocation strategy:

How It Works

- The process scans memory blocks from the beginning.
- The first block that is **large enough** is allocated.
- This is fast but may leave small unusable holes.

Example

- Blocks: [100, 500, 200, 300, 600]
- Process: 212
- Allocation: 500 (first block that fits)

Code Concept

```
Python

for block in blocks:
    if block >= process_size:
        allocate and break
```

3.3 Analysis

Strategy	Speed	Efficiency	Fragmentation
First Fit	Fast	Medium	High
Best Fit	Moderate	High	Low

3.4 Best Fit Allocation

Best Fit attempts to minimize wasted space:

How It Works

- Scans all memory blocks.
- Selects the **smallest block** that fits the process.
- Reduces internal fragmentation.

Example

- Blocks: [100, 500, 200, 300, 600]
- Process: 212
- Allocation: 300 (best size match)

Code Concept

```
Python

CopyEdit
best_block = find_smallest_fitting_block()
```

3.4 Visualization

To demonstrate how memory blocks are allocated, the project uses **bar charts**. Each bar represents a block, and colored sections show which process got which block using First Fit or Best Fit strategy.

Visualization Benefits

- Easy to compare allocations
- Shows fragmentation clearly
- Helps understand memory efficiency visually

Deadlock Detection

How the project detects **deadlocks** in a system of processes and resources. It includes the logic, algorithm used, and a practical example with outputs.

4.1 What is a Deadlock?

A **deadlock** occurs when a group of processes are waiting on each other to release resources, and none of them can proceed. It leads to system halt unless resolved.

4.2 Conditions for Deadlock

A deadlock arises when **all four** of these conditions hold:

1. **Mutual Exclusion** – Only one process can hold a resource.
2. **Hold and Wait** – A process holding one resource is waiting for another.
3. **No Preemption** – Resources can't be forcibly taken from a process.
4. **Circular Wait** – A set of processes form a cycle, each waiting for the next.

4.3 Deadlock Detection Algorithm Used

This project uses a variation of the **Banker's Algorithm** for detecting deadlocks.

Inputs Required:

- Allocation Matrix
- Max Demand Matrix
- Available Resources

Steps:

1. Calculate the **Need Matrix**: `Need = Max - Allocation`
2. Find a process whose needs can be satisfied with available resources.
3. If found, assume it finishes and release its resources.
4. Repeat until all processes are marked "safe" or no further progress is possible.

4.4 Output Results

The program returns:

- **Safe Sequence** (if no deadlock)
- **List of Deadlocked Processes** (if deadlock exists)

Example Output

```
bash
CopyEdit
System is in Deadlock!
Deadlocked Processes: [P2, P4]
```

OR

```
bash
CopyEdit
System is Safe.
Safe Sequence: [P0, P2, P3, P1]
```

4.5 Code Snippet

```
python
CopyEdit
need[i][j] = max[i][j] - alloc[i][j]

# Check for a process that can execute
if all(need[i][j] <= available[j] for j in range(n)):
    work += allocation[i]
    mark process as finished
```

4.6 Importance of Detection

- Helps avoid system hang
- Allows recovery from unsafe states
- Enhances system reliability

Scheduling Algorithms

Two CPU scheduling algorithms implemented in the project: **Shortest Job First (SJF)** and **Round Robin (RR)**. It compares their logic, execution flow, and performance metrics such as waiting time, turnaround time, context switches, CPU utilization, and throughput.

5.1 Overview of CPU Scheduling

CPU scheduling determines which process runs at a given time when multiple processes are ready to execute. The goal is to maximize CPU efficiency and reduce waiting time, turnaround time, etc.

5.2 Shortest Job First (SJF)

Working:

SJF selects the process with the shortest burst time that has arrived. It is **non-preemptive**, so once a process starts, it runs to completion.

Key Points:

- Selects the shortest burst time job among arrived processes
- Non-preemptive
- Ideal for batch systems

Metrics Calculated:

- Average Waiting Time
- Average Turnaround Time
- Context Switches
- CPU Utilization
- Throughput

Pros:

- Minimizes average waiting time

Cons:

- Not suitable for real-time systems
- Can lead to starvation of long processes

5.3 Round Robin (RR)

Working:

RR assigns a fixed time quantum to each process in a circular queue. If a process doesn't finish in its time slice, it is moved to the end of the queue.

Key Points:

- Preemptive scheduling
- Suitable for time-sharing systems
- Fair time distribution among processes

Metrics Calculated:

- Average Waiting Time
- Average Turnaround Time
- Context Switches
- CPU Utilization
- Throughput

Pros:

- Fair and responsive
- Ideal for interactive systems

Cons:

- High context switching overhead
- Performance depends on quantum size

5.5 Performance Metrics

Metric	Description
Avg. Waiting Time	Time a process waits before execution
Avg. Turnaround Time	Total time taken from arrival to completion
Context Switches	Number of times CPU switches between processes
CPU Utilization	Ratio of time CPU is actively executing vs total time
Throughput	Number of processes completed per unit time

5.6 Comparison Table

Metric	SJF	Round Robin
Avg. Waiting Time	Lower	Higher (due to preemption)
Turnaround Time	Lower	Slightly higher
Context Switches	Fewer	More frequent
CPU Utilization	Moderate to High	High (depends on quantum)

Memory Management

The implementation of memory allocation strategies used in the project, specifically **First Fit** and **Best Fit** algorithms. It also includes visualization of memory block allocation using bar graphs to make the process easier to understand.

6.1 Overview of Memory Management

Memory management involves allocating memory blocks to processes efficiently. The aim is to:

- Minimize fragmentation
- Maximize memory usage
- Ensure fair and fast allocation

6.2 First Fit Algorithm

Working:

- Allocates the first available memory block that is large enough to hold the process.
- Scans memory blocks from the beginning.

Example:

Given blocks: `100 500 200 300 600`
Process: `212`
→ Allocated to block `500` (first block ≥ 212)

Advantages:

- Fast and simple
- Lower overhead

Disadvantages:

- Leads to external fragmentation
- May leave many small unusable holes

6.3 Best Fit Algorithm

Working:

- Allocates the **smallest** block that is **sufficiently large** for the process.
- Searches entire list before allocation.

Example:

Given blocks: `100 500 200 300 600`
Process: `212`
→ Allocated to block `300` (smallest fit ≥ 212)

Advantages:

- Reduces wasted memory
- More memory-efficient

Disadvantages:

- Slower than First Fit
- May still cause fragmentation

6.4 Sample Input (During Execution)

```
Enter memory block sizes: 100 500 200 300 600
Enter process sizes: 212 417 112 426
```

6.5 Visualization

Memory allocations for both algorithms are visualized using bar graphs with `matplotlib` (from `memory.py`):

- Each bar represents a block and how it is filled by processes.
- Helps identify fragmentation and efficiency.

Benefits of Visualization:

- Clear view of memory usage
- Easier algorithm comparison
- Visual proof of space wastage or efficiency

6.6 Comparison Table

Feature	First Fit	Best Fit
Speed	Faster	Slower (full scan)
Memory Utilization	Moderate	Higher
Fragmentation	Higher	Lower (usually)
Implementation	Easier	Slightly complex

Performance Analysis & Comparison

Detailed comparison of both scheduling algorithms — **Shortest Job First (SJF)** and **Round Robin (RR)** — based on various performance metrics.

7.1 Metrics Analyzed

- **Average Waiting Time**
- **Average Turnaround Time**
- **Context Switches**
- **CPU Utilization**
- **Throughput**

7.2 Tabular Comparison

Metric	SJF	Round Robin
Average Waiting Time	e.g., 4.25 ms	e.g., 6.75 ms
Average Turnaround Time	e.g., 9.00 ms	e.g., 10.50 ms
Context Switches	e.g., 4	e.g., 7
CPU Utilization (%)	e.g., 95.0%	e.g., 92.3%
Throughput (jobs/sec)	e.g., 0.35	e.g., 0.33

Note :

(Values will depend on your input and results; use actual output from your implementation.)

7.3 Graphical Comparison

A bar graph is plotted for visual comparison across the five metrics:

- **X-axis**: Scheduling Algorithms (SJF, RR)
- **Y-axis**: Metric values
- **Bars**: Color-coded for each metric

The graph is generated using **`matplotlib`** in **`plot.py`.**

7.4 Observations

- **SJF** performs better in terms of **waiting time** and **turnaround time** but has the drawback of being **non-preemptive**.
- **Round Robin** is fairer in a **multi-user environment** but may cause **more context switches** and slightly lower CPU efficiency.
- Throughput and CPU utilization show **minor differences**, depending on the time quantum.

7.5 Conclusion

- Choose **SJF** when you want **high efficiency** and have **batch jobs**.
- Use **RR** when you need **fairness** and are handling **interactive processes**.

Conclusion & Future Work

8.1 Summary of Learnings

Provided hands-on experience with fundamental **Operating System concepts**, especially in:

- **CPU Scheduling Algorithms** – Implemented and compared **SJF** and **Round Robin** based on real-time metrics like turnaround time, waiting time, context switches, CPU utilization, and throughput.
- **Memory Management** – Integrated **First Fit** and **Best Fit** algorithms with visual bar graphs to demonstrate dynamic memory allocation.
- **Deadlock Detection** – Implemented detection mechanisms using Resource Allocation Graphs and safe sequence validation.

These implementations enhanced the understanding of theoretical concepts and their practical behavior in real systems.

8.2 Challenges Faced

- Designing a fair comparison mechanism for SJF (non-preemptive) and RR (preemptive).
- Handling simultaneous process arrivals and dynamic memory allocation.
- Visualizing performance metrics in an informative and user-friendly format.

8.3 Future Enhancements

This project lays a strong foundation for further development. In future versions, we — **Saif**, **Fahad**, and **Zeeshan** — plan to enhance the system by introducing:

- **Preemptive SJF (Shortest Remaining Time First)** to better reflect time-sharing systems.
- **Advanced Visualization** using libraries like `seaborn` or `Plotly` for dynamic, interactive graphs.
- **Graphical User Interface (GUI)** with tools like Tkinter or PyQt to allow users to simulate scheduling and memory allocation through a user-friendly interface.
- **Additional Scheduling Algorithms** such as Priority Scheduling, Multilevel Queue, and Multilevel Feedback Queue for broader comparison.
- **Paging and Segmentation Simulation** to better model real memory management techniques.
- **Multithreading/Multiprocessing Support** to implement true concurrency and shared memory IPC more effectively.

8.4 Final Thoughts

This project not only strengthened understanding of **core OS mechanisms**, but also demonstrated how complex system-level decisions can be modeled and visualized through code. The balance between fairness, efficiency, and system performance is critical in designing real-time operating systems.