Table of Contents

[**Title:** 4](#_Toc200671443)

[**1. Introduction** 4](#_Toc200671444)

[**2. Objectives** 4](#_Toc200671445)

[**3. System Overview** 4](#_Toc200671446)

[3.1 Page Replacement Visualizer 4](#_Toc200671447)

[3.2 CPU Scheduling Simulator 5](#_Toc200671448)

[3.3 Memory Management Visualizer 5](#_Toc200671449)

[3.4 Deadlock Handling Module 5](#_Toc200671450)

[3.5 Shell Command Helper 5](#_Toc200671451)

[**4. Features** 5](#_Toc200671452)

[**5. Challenges** 6](#_Toc200671453)

[**6. Code Snippets** 6](#_Toc200671454)

[6.1 main.py 6](#_Toc200671455)

[**6.1.1: Menu-Driven Main Program** 6](#_Toc200671456)

[Explanation: 6](#_Toc200671457)

[6.1.2: Page Replacement Runner 6](#_Toc200671458)

[Explanation: 7](#_Toc200671459)

[6.1.3: Deadlock Input Collection 7](#_Toc200671460)

[Explanation: 7](#_Toc200671461)

[6.2 Simulation of Page Replacement 8](#_Toc200671462)

[6.2.1: FIFO.PY 8](#_Toc200671463)

[Output: 9](#_Toc200671464)

[Graph: 9](#_Toc200671465)

[Gant chart: 10](#_Toc200671466)

[6.2.2: LRU.PY 10](#_Toc200671467)

[Output: 11](#_Toc200671468)

[Graph: 11](#_Toc200671469)

[Gantt chart: 12](#_Toc200671470)

[6.2.3: OPTIMAL.PY 12](#_Toc200671471)

[Output: 13](#_Toc200671472)

[Graph: 14](#_Toc200671473)

[Gantt chart: 14](#_Toc200671474)

[**6.3 Scheduling** 14](#_Toc200671475)

[6.3.1: FCFS.PY 14](#_Toc200671476)

[Output:- 16](#_Toc200671477)

[Gantt chart: 16](#_Toc200671478)

[6.3.2: SJF.PY 16](#_Toc200671479)

[Output: 17](#_Toc200671480)

[Gantt chart: 18](#_Toc200671481)

[6.3.4: ROUND ROBIN.PY 18](#_Toc200671482)

[Output: 20](#_Toc200671483)

[‘ 20](#_Toc200671484)

[Gantt chart: 21](#_Toc200671485)

[**6.4 Memory Management** 21](#_Toc200671486)

[6.4.1: First fit.py 21](#_Toc200671487)

[Output: 22](#_Toc200671488)

[Graph: 22](#_Toc200671489)

[6.4.2: Best fit.py 23](#_Toc200671490)

[Output: 24](#_Toc200671491)

[Graph: 24](#_Toc200671492)

[**6.5 Deadlock** 24](#_Toc200671493)

[6.5.1: Detection.py 24](#_Toc200671494)

[Output: 26](#_Toc200671495)

[Graph: 26](#_Toc200671496)

[**7. Future Improvements / Conclusion** 27](#_Toc200671497)

[**Future Improvements:** 27](#_Toc200671498)

[**Conclusion:** 27](#_Toc200671499)

Table of Figure:

[Figure 1:Deadlock Input Collection 7](#_Toc200671393)

[Figure 2:FIFO.PY 9](#_Toc200671394)

[Figure 3:FIFO.PY Graph 9](#_Toc200671395)

[Figure 4:FIFO.PY Gant Chart 10](#_Toc200671396)

[Figure 5:LRU.PY 11](#_Toc200671397)

[Figure 6: LRU.PY Graph 11](#_Toc200671398)

[Figure 7:LRU.PY Gantt Chart 12](#_Toc200671399)

[Figure 8: OPTIMAL.PY 13](#_Toc200671400)

[Figure 9: OPTIMAL.PY Graph 14](#_Toc200671401)

[Figure 10: OPTIMAL.PY Gant Chart 14](#_Toc200671402)

[Figure 11: FCFS.PY 16](#_Toc200671403)

[Figure 12:FCFS.PY Gantt Chart 16](#_Toc200671404)

[Figure 13: SJF.PY 18](#_Toc200671405)

[Figure 14:SJF.PY Gantt Chart 18](#_Toc200671406)

[Figure 15:ROUND ROBIN.PY 20](#_Toc200671407)

[Figure 16:ROUND ROBIN.PY Gantt chart 21](#_Toc200671408)

[Figure 17: First fit.py 22](#_Toc200671409)

[Figure 18:First fit.py Graph 22](#_Toc200671410)

[Figure 19:Best fit.py 24](#_Toc200671411)

[Figure 20: Best fit.py Graph 24](#_Toc200671412)

[Figure 21:Detection.py 26](#_Toc200671413)

[Figure 22:Detection.py Graph 26](#_Toc200671414)

**Title:**

**OS Algorithm Visualizer & Simulator**  
*(Simulation of Page Replacement, Scheduling, Memory Management, Deadlock Handling, and Shell Commands)*

# **1. Introduction**

Operating systems (OS) play a vital role in managing hardware and software resources. Key areas like page replacement, CPU scheduling, memory management, and deadlock handling are essential to OS design. However, these concepts can be difficult to understand through theory alone.

This project aims to provide an **interactive visual platform** to simulate and understand these algorithms in an educational and engaging manner.

# **2. Objectives**

The main objectives of the project are:

* To simulate and compare **page replacement algorithms** such as FIFO, LRU, and Optimal.
* To visualize and evaluate **CPU scheduling algorithms** including FCFS, SJF, and Round Robin.
* To demonstrate **memory allocation strategies** like First Fit, Best Fit, and Worst Fit.
* To simulate **deadlock detection and handling** techniques.
* To assist learners in understanding **Linux shell commands and scripting** with practical examples and outputs.
* To make learning **interactive and visually intuitive** using tables, graphs, and step-by-step animations.

# **3. System Overview**

The system is composed of multiple independent but interconnected modules, each responsible for simulating a specific OS concept:

### 3.1 Page Replacement Visualizer

* **Input:** Page reference string, number of frames
* **Output:** Frame states over time, page hits/misses, comparison chart
* **Algorithms:** FIFO, LRU, Optimal

### 3.2 CPU Scheduling Simulator

* **Input:** Process details (arrival time, burst time, quantum)
* **Output:** Gantt chart, average waiting/turnaround time
* **Algorithms:** FCFS, SJF (preemptive/non-preemptive), Round Robin

### 3.3 Memory Management Visualizer

* **Input:** Process memory sizes, available memory blocks
* **Output:** Allocation results and fragmentation details
* **Algorithms:** First Fit, Best Fit

### 3.4 Deadlock Handling Module

* **Input:** Allocation and request matrices
* **Output:** Safe sequence or deadlock detection
* **Techniques:** detector Algorithm, Resource Allocation Graph (RAG)

### 3.5 Shell Command Helper

* **Input:** User-selected shell commands
* **Output:** Syntax help, real-time execution result
* **Examples:** ls, pwd, grep, awk, basic scripts

# **4. Features**

| **Module** | **Description** |
| --- | --- |
| * Page Replacement | * Simulates FIFO, LRU, and Optimal algorithms visually. Shows frame changes and hit/miss count. |
| * CPU Scheduling | * Visualizes FCFS, SJF, and RR with Gantt charts and average time calculations. |
| * Memory Management | * Allocates memory using First Fit, Best Fit, Displays memory map. |
| * Deadlock Handling | * Simulates safe sequence using Banker's Algorithm. Shows wait-for graphs. |
| * Shell Scripting | * Lists and explains common Linux shell commands. Executes basic scripts. |
| * Visual Interface | * Graphs, tables, animations, and tooltips for enhanced learning. |

# **5. Challenges**

* Developing dynamic and **efficient visualizations** for complex algorithms.
* Handling **varied user inputs** and preventing incorrect data submission.
* Creating a **responsive user interface** compatible across screen sizes.
* Simulating **deadlock detection** with accurate matrix and graph logic.
* Providing **safe shell execution** without security risks.

# **6. Code Snippets**

### 6.1 main.py

**6.1.1: Menu-Driven Main Program**

def main():

while True:

print\_menu()

choice = input("Select an option (1-5): ").strip()

if choice == '1':

pages = get\_pages\_from\_user()

run\_page\_replacement\_demo(pages)

elif choice == '2':

run\_scheduling\_demo()

elif choice == '3':

run\_memory\_management\_demo()

elif choice == '4':

run\_deadlock\_demo()

elif choice == '5':

print("Exiting. Goodbye!")

sys.exit(0)

else:

print("Invalid choice. Please enter a number between 1 and 5.")

Explanation:  
This is the **main control loop** for the whole OS Concepts demo. It repeatedly displays a menu, takes user input, and runs the corresponding module like page replacement, scheduling, memory management, or deadlock detection.

### 6.1.2: Page Replacement Runner

def run\_page\_replacement\_demo(pages):

capacity = int(input("Enter frame capacity: "))

fifo = FIFO(capacity)

for i, page in enumerate(pages):

fifo.access\_page(page, i)

faults, hits, frames = fifo.get\_results()

visualize\_page\_replacement("FIFO", faults, hits, frames, pages)

lru = LRU(capacity)

for i, page in enumerate(pages):

lru.access\_page(page, i)

faults, hits, frames = lru.get\_results()

visualize\_page\_replacement("LRU", faults, hits, frames, pages)

optimal = Optimal(capacity)

optimal.run\_simulation(pages)

faults, hits, frames = optimal.get\_results()

visualize\_page\_replacement("Optimal", faults, hits, frames, pages)

Explanation:  
This function runs **3 page replacement algorithms** — FIFO, LRU, and Optimal — using the user-provided pages and frame capacity. It visualizes the result for each algorithm, showing page hits/faults.

### 6.1.3: Deadlock Input Collection

def get\_deadlock\_input():

r = int(input("Enter number of resource types: "))

p = int(input("Enter number of processes: "))

resource\_names = [f'R{i+1}' for i in range(r)]

processes = [f'P{i+1}' for i in range(p)]

print("\nEnter total instances of each resource:")

total\_resources = [int(input(f" Total {name}: ")) for name in resource\_names]

print("\nEnter Allocation matrix:")

allocation = []

for i in range(p):

row = list(map(int, input(f" Allocation for {processes[i]}: ").split()))

allocation.append(row)

print("\nEnter Request matrix:")

request = []

for i in range(p):

row = list(map(int, input(f" Request for {processes[i]}: ").split()))

request.append(row)

return processes, resource\_names, total\_resources, allocation, request

Explanation:  
This helper function collects input for **deadlock detection**, including:

* total number of resources and processes,
* available resources,
* current allocations,
* and pending requests.

It returns all required matrices and labels to be used in detection and visualization.

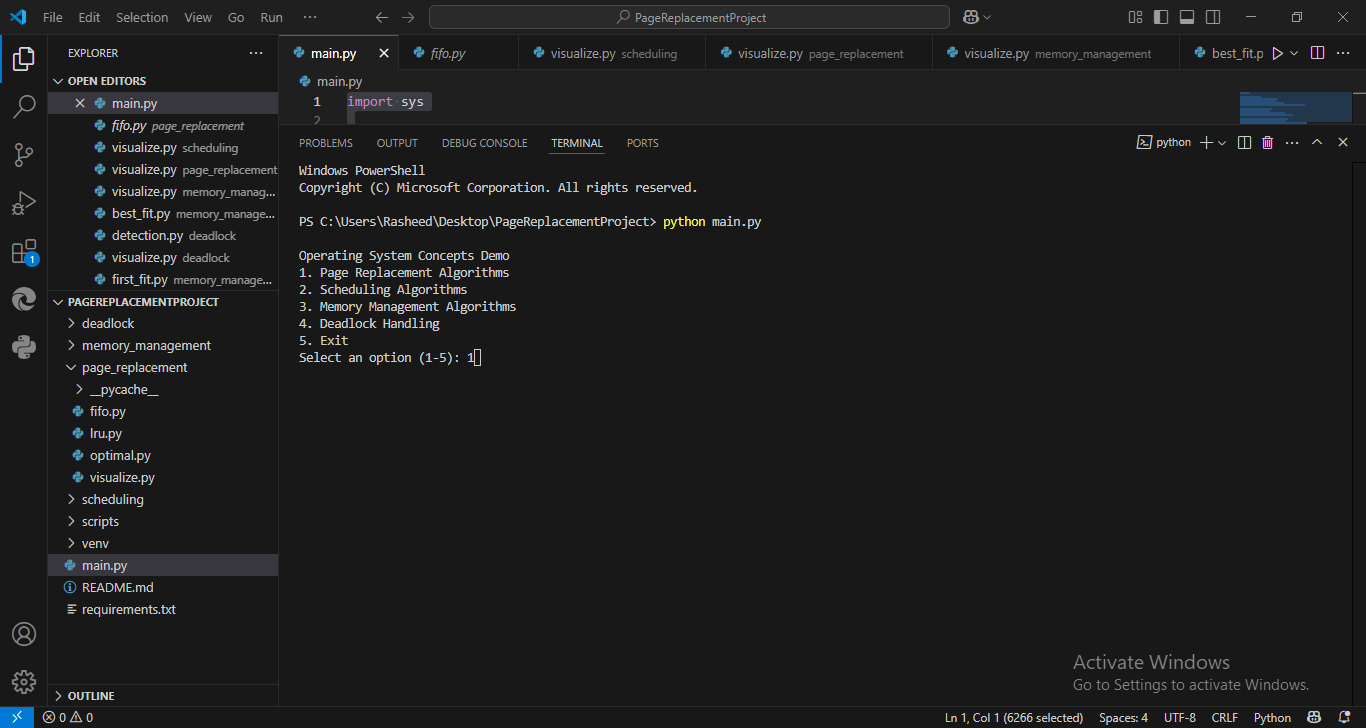


Figure 1:Deadlock Input Collection

### 6.2 Simulation of Page Replacement

### 6.2.1: FIFO.PY

The FIFO class simulates the **First-In-First-Out page replacement algorithm**. In the constructor:

def \_\_init\_\_(self, capacity):

self.capacity = capacity

self.queue = []

self.page\_faults = 0

self.page\_hits = 0

self.steps = []

* It sets the memory size (capacity), an empty list for the current pages (queue), counters for page faults and hits, and a steps list to track each operation.

The access\_page method handles one page access:

def access\_page(self, page, step):

status = ""

if page in self.queue:

self.page\_hits += 1

status = "Hit"

else:

self.page\_faults += 1

status = "Fault"

if len(self.queue) == self.capacity:

self.queue.pop(0)

self.queue.append(page)

self.steps.append((step, page, self.queue[:], status))

* If the page is already in memory, it counts as a hit. If not, it's a fault. If memory is full, the oldest page is removed. The new page is added, and the action is logged.

The get\_results method returns the final stats:

def get\_results(self):

return self.page\_faults, self.page\_hits, self.steps

* It provides total faults, hits, and the detailed access history for reporting.

### Output:

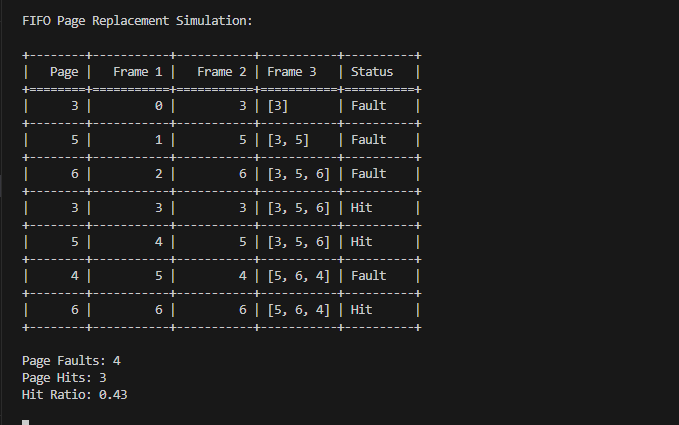


Figure 2:FIFO.PY

### Graph:

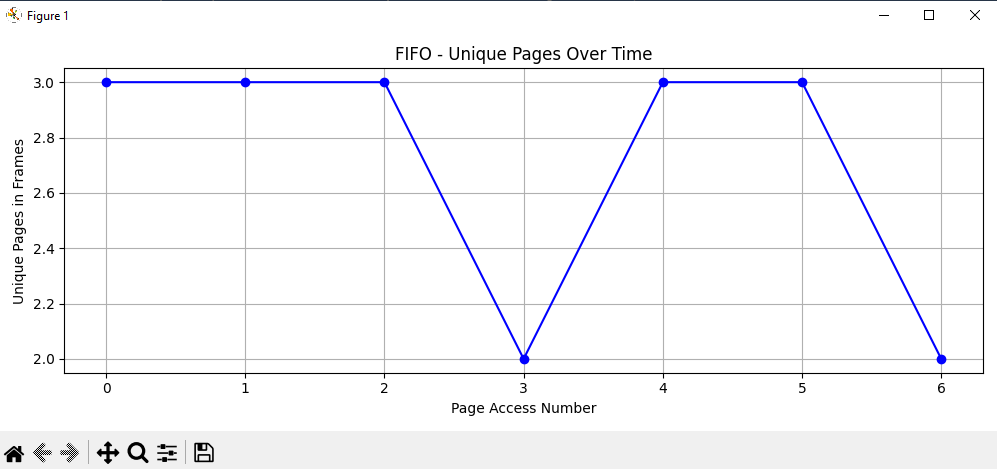


Figure 3:FIFO.PY Graph

### Gant chart:

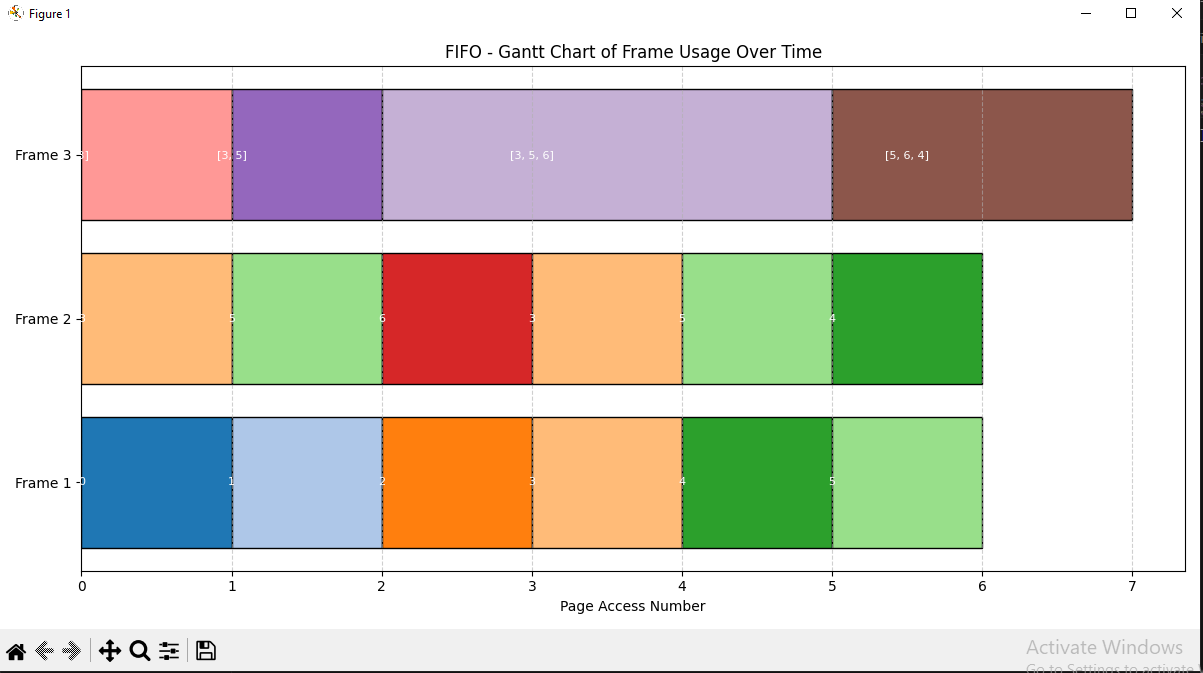


Figure 4:FIFO.PY Gant Chart

### 6.2.2: LRU.PY

def \_\_init\_\_(self, capacity):

self.capacity = capacity

self.queue = [] # Frames (most recent at end)

self.page\_faults = 0

self.page\_hits = 0

self.steps = [] # (step, page, frame, status)

* Initializes the LRU object with a given capacity. queue tracks pages in memory. page\_faults and page\_hits count the results, and steps stores each operation for visualization.

def access\_page(self, page, step):

status = ""

if page in self.queue:

self.page\_hits += 1

self.queue.remove(page)

self.queue.append(page)

status = "Hit"

else:

self.page\_faults += 1

if len(self.queue) == self.capacity:

self.queue.pop(0)

self.queue.append(page)

status = "Fault"

self.steps.append((step, page, self.queue[:], status))

* Handles accessing a page. If the page is found (hit), it's moved to the end. If not (fault), the oldest page is removed (if full), and the new page is added.

def get\_results(self):

return self.page\_faults, self.page\_hits, self.steps

* Returns the final number of page faults, page hits, and the list of all access steps for later display or analysis.

### Output:

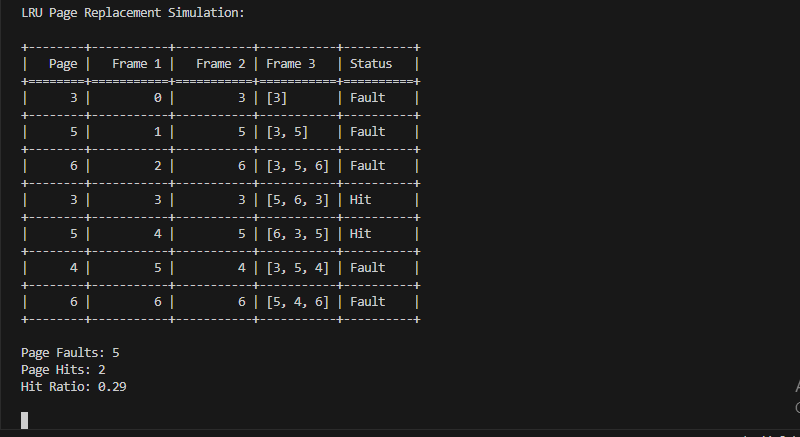


Figure 5:LRU.PY

### Graph:

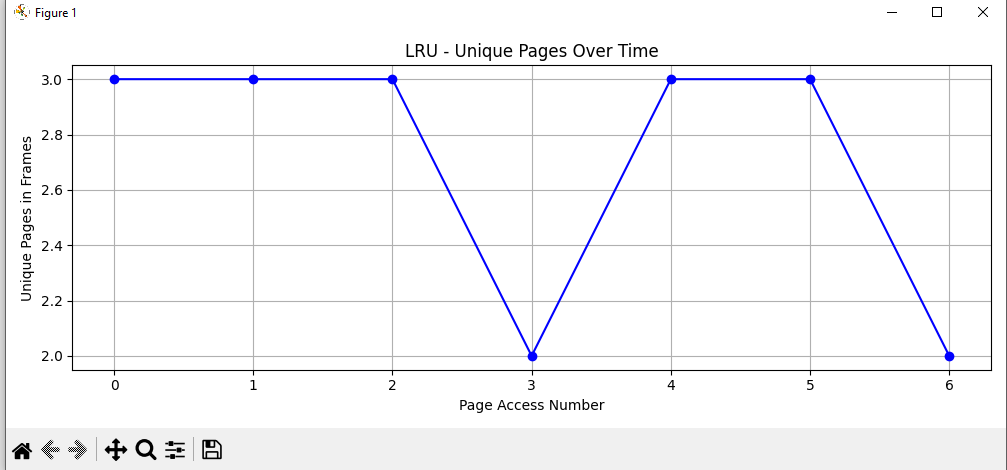


Figure 6: LRU.PY Graph

### Gantt chart:

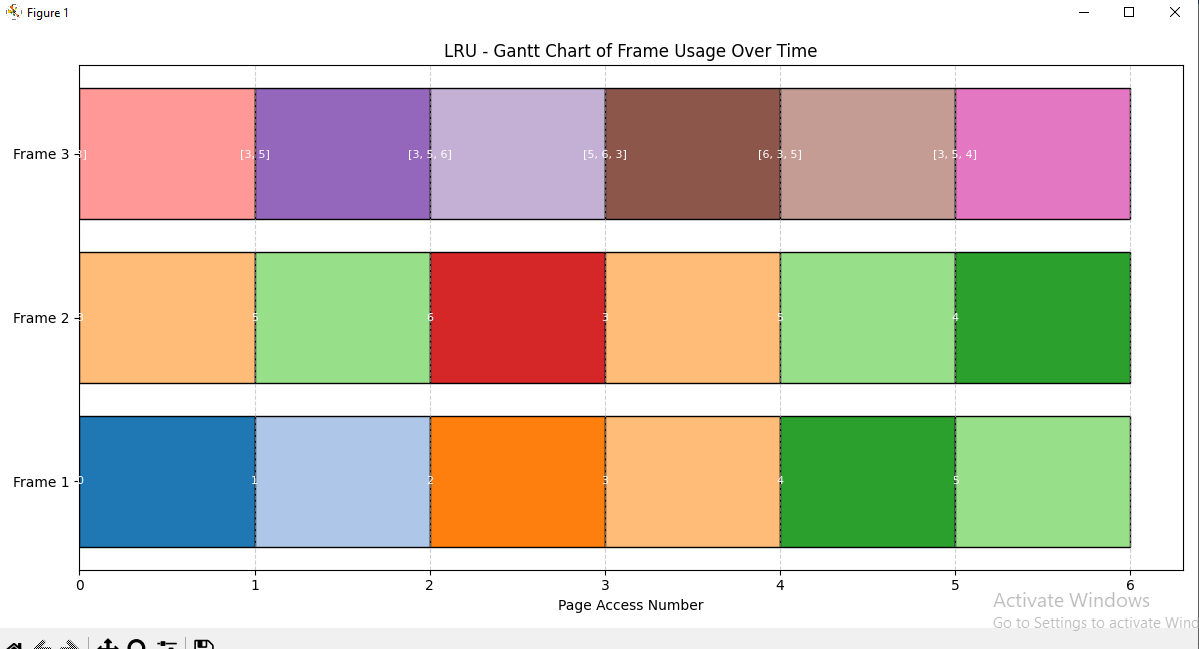


Figure 7:LRU.PY Gantt Chart

### 6.2.3: OPTIMAL.PY

def \_\_init\_\_(self, capacity):

self.capacity = capacity

self.frames = []

self.page\_faults = 0

self.page\_hits = 0

self.steps = []

* **Purpose**: Initialize variables like capacity, list of frames, counters for hits/faults, and step tracking.

def predict(self, pages, current\_index):

farthest = current\_index

page\_to\_replace = None

for page in self.frames:

try:

next\_use = pages[current\_index + 1:].index(page) + current\_index + 1

except ValueError:

return page # If page is not used again, it's best to replace

if next\_use > farthest:

farthest = next\_use

page\_to\_replace = page

return page\_to\_replace or self.frames[0]

* **Purpose**: Predict the page to be replaced by looking **farthest into the future**.
* **Logic**:
  + If a page is **never used again**, replace it.
  + Otherwise, replace the one that will be used **latest in the future**.

def access\_page(self, page, pages, step):

status = ""

if page in self.frames:

self.page\_hits += 1

status = "Hit"

else:

self.page\_faults += 1

if len(self.frames) == self.capacity:

replace = self.predict(pages, step)

self.frames[self.frames.index(replace)] = page

else:

self.frames.append(page)

status = "Fault"

self.steps.append((step, page, self.frames[:], status))

* **Purpose**: Access a page and decide whether it’s a **hit or fault**.
* **Logic**:
  + **Hit**: Page is already in memory.
  + **Fault**: Page not in memory → add or replace one using predict().
  + Track every step including the frame state.

def run\_simulation(self, pages):

for i, page in enumerate(pages):

self.access\_page(page, pages, i)

* **Purpose**: Loop through all pages and call access\_page().

def get\_results(self):

return self.page\_faults, self.page\_hits, self.steps

* **Purpose**: Return the total **page faults**, **hits**, and **step details** (for analysis or printing).

### Output:

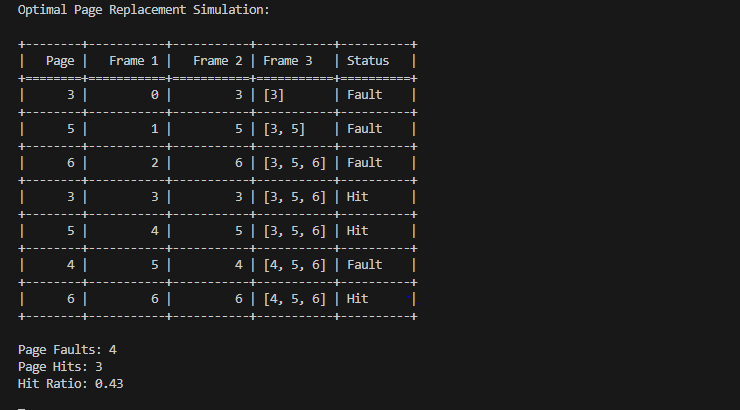


Figure 8: OPTIMAL.PY

### Graph:

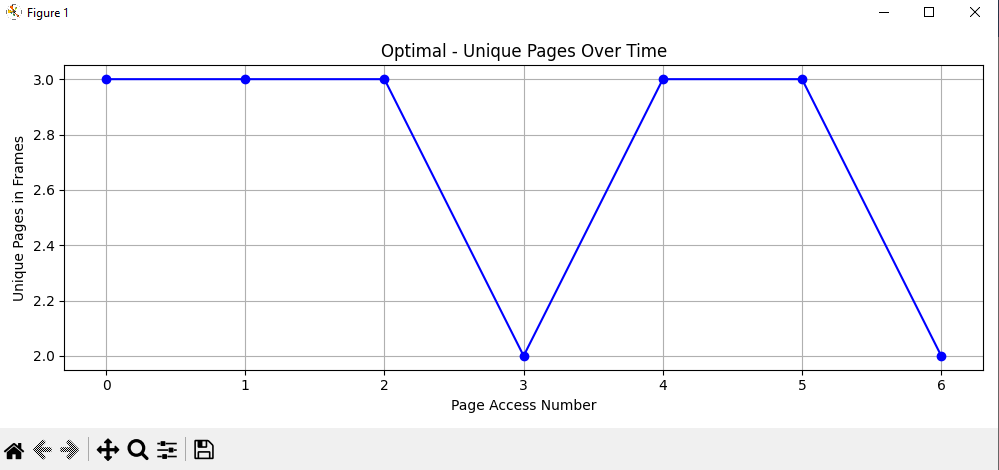


Figure 9: OPTIMAL.PY Graph

### Gantt chart:

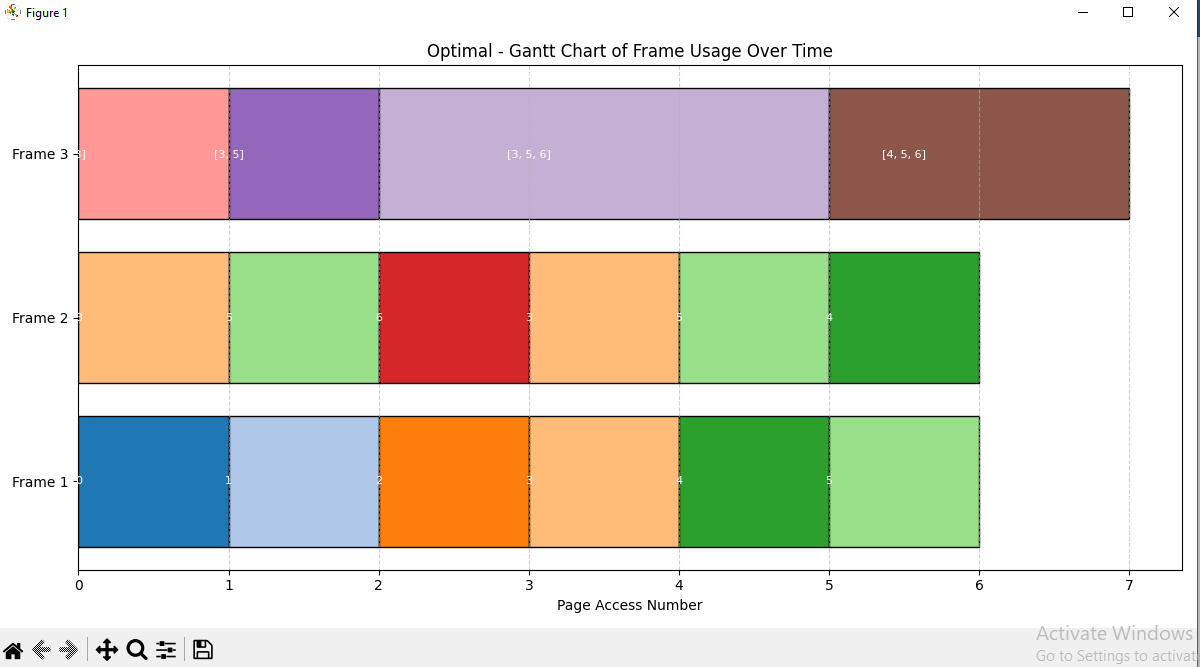


Figure 10: OPTIMAL.PY Gant Chart

# **6.3 Scheduling**

### 6.3.1: FCFS.PY

def \_\_init\_\_(self, processes):

self.processes = sorted(processes, key=lambda x: x['arrival\_time']) # Sort by arrival time

self.completion\_times = {}

self.waiting\_times = {}

self.turnaround\_times = {}

* processes: A list of dictionaries with each process info:

{'name': 'P1', 'arrival\_time': 0, 'burst\_time': 5}

* Sorts processes based on arrival\_time (FCFS rule).
* Initializes dictionaries to store:
  + **completion time**
  + **waiting time**
  + **turnaround time**

def run(self):

current\_time = 0

for p in self.processes:

arrival = p['arrival\_time']

if current\_time < arrival:

current\_time = arrival # CPU waits for the process to arrive

start\_time = current\_time

complete = start\_time + p['burst\_time']

self.completion\_times[p['name']] = complete

self.turnaround\_times[p['name']] = complete - arrival

self.waiting\_times[p['name']] = start\_time - arrival

current\_time = complete

* For each process:
  + Wait if the process hasn’t arrived (current\_time < arrival).
  + Set start\_time as the current time.
  + Calculate:
    - completion\_time = start\_time + burst\_time
    - turnaround\_time = completion\_time - arrival\_time
    - waiting\_time = start\_time - arrival\_time
  + Move current\_time to after the current process finishes.

def get\_results(self):

return self.completion\_times, self.waiting\_times, self.turnaround\_times

* Returns all three dictionaries:
  + completion\_times: When each process finished.
  + waiting\_times: How long each process waited before starting.
  + turnaround\_times: Total time from arrival to completion

### Output:-

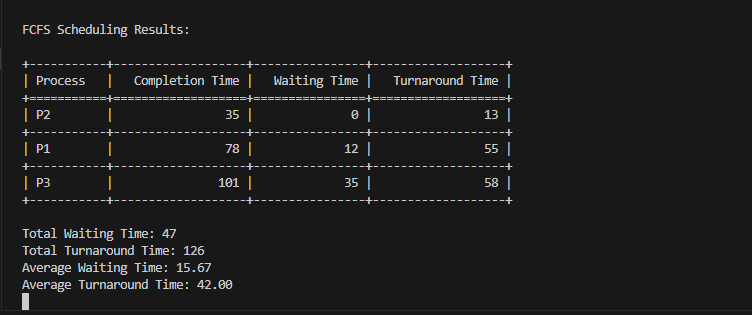


Figure 11: FCFS.PY

### Gantt chart:

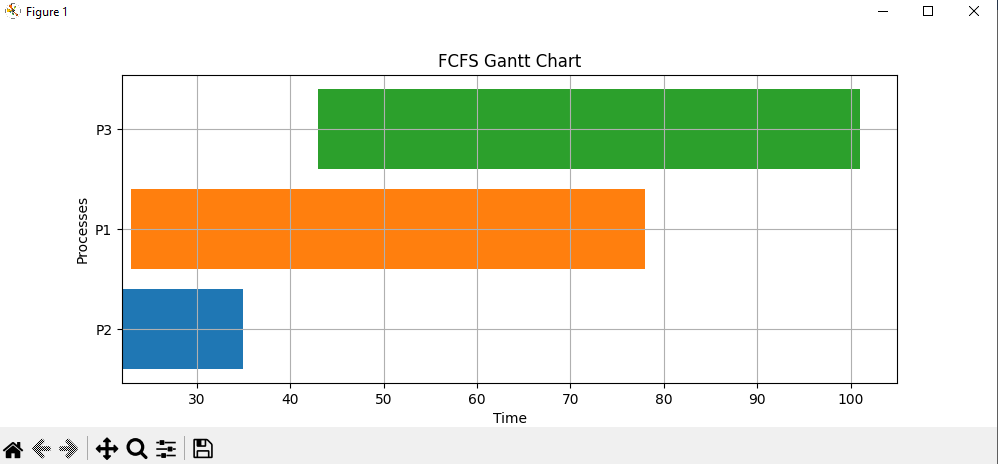


Figure 12:FCFS.PY Gantt Chart

### 6.3.2: SJF.PY

def \_\_init\_\_(self, processes):

self.processes = processes

self.completion\_times = {}

self.waiting\_times = {}

self.turnaround\_times = {}

* processes: List of dictionaries, each with 'name', 'arrival\_time', and 'burst\_time'.
* Initializes dictionaries to store:
  + Completion times
  + Waiting times
  + Turnaround times

def run(self):

processes = self.processes[:]

time = 0

completed = []

while len(completed) < len(processes):

available = [p for p in processes if p['arrival\_time'] <= time and p['name'] not in completed]

if not available:

time += 1

continue

next\_process = min(available, key=lambda x: x['burst\_time'])

start = time

time += next\_process['burst\_time']

name = next\_process['name']

self.completion\_times[name] = time

self.turnaround\_times[name] = time - next\_process['arrival\_time']

self.waiting\_times[name] = self.turnaround\_times[name] - next\_process['burst\_time']

completed.append(name)

* Makes a copy of processes.
* Tracks time (time) and completed process names (completed).
* At each time unit:
  + Filters **available** processes (already arrived and not yet completed).
  + If no process is available, time advances (time += 1).
  + Otherwise, selects the one with the **shortest burst time**.
* Computes:
  + completion\_time = current time + burst time
  + turnaround\_time = completion\_time - arrival\_time
  + waiting\_time = turnaround\_time - burst\_time
* Adds process name to the completed list.

def get\_results(self):

return self.completion\_times, self.waiting\_times, self.turnaround\_times

* Returns all calculated metrics for each process.

### Output:

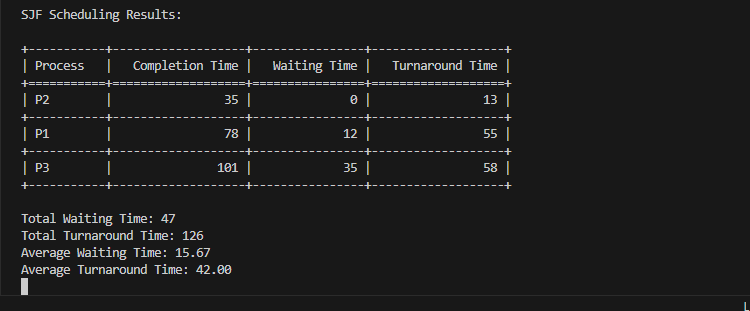


Figure 13: SJF.PY

### Gantt chart:

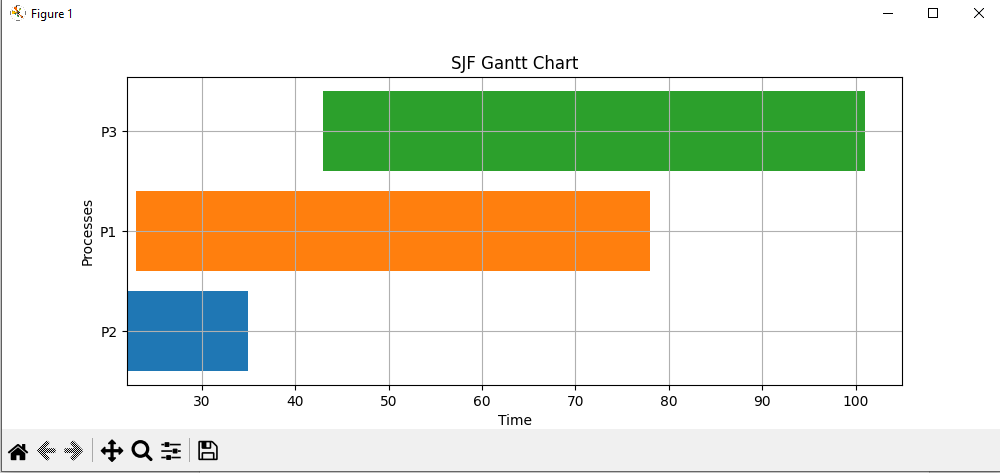


Figure 14:SJF.PY Gantt Chart

### 6.3.4: ROUND ROBIN.PY

class RoundRobin:

def \_\_init\_\_(self, processes, time\_quantum):

self.processes = processes

self.time\_quantum = time\_quantum

self.completion\_times = {}

self.waiting\_times = {}

self.turnaround\_times = {}

* processes: List of process dictionaries with name, arrival\_time, and burst\_time.
* time\_quantum: The fixed CPU time slice per process.
* Initializes dictionaries for:
  + Completion times
  + Waiting times
  + Turnaround times

def run(self):

time = 0

queue = []

remaining\_bt = {p['name']: p['burst\_time'] for p in self.processes}

arrival\_dict = {p['name']: p['arrival\_time'] for p in self.processes}

visited = {p['name']: False for p in self.processes}

* time: Keeps track of the current time.
* queue: Holds the ready processes.
* remaining\_bt: Remaining burst time per process.
* arrival\_dict: Maps process name to its arrival time.
* visited: Ensures a process is only added to the queue once on arrival.

while len(self.completion\_times) < len(self.processes):

# Add newly arrived processes to the queue

for p in self.processes:

if p['arrival\_time'] <= time and not visited[p['name']]:

queue.append(p['name'])

visited[p['name']] = True

if not queue:

time += 1

continue

* Adds processes to the queue as they arrive (and only once).
* If no process is ready, time increments (CPU idle).

current = queue.pop(0)

exec\_time = min(self.time\_quantum, remaining\_bt[current])

time += exec\_time

remaining\_bt[current] -= exec\_time

* Picks the first process in the queue (FIFO).
* Executes it for min(time\_quantum, remaining\_bt) time units.
* Decreases remaining burst time.

for p in self.processes:

if p['arrival\_time'] <= time and not visited[p['name']]:

queue.append(p['name'])

visited[p['name']] = True

* After execution, new processes might have arrived → they are added.

if remaining\_bt[current] > 0:

queue.append(current)

else:

self.completion\_times[current] = time

burst = next(p['burst\_time'] for p in self.processes if p['name'] == current)

arrival = arrival\_dict[current]

self.turnaround\_times[current] = time - arrival

self.waiting\_times[current] = self.turnaround\_times[current] - burst

* If not finished, re-queue the process.
* If finished:
  + Save its **completion time**.
  + Calculate:
    - **Turnaround time** = completion - arrival
    - **Waiting time** = turnaround - burst

def get\_results(self):

return self.completion\_times, self.waiting\_times, self.turnaround\_times

Returns computed results.

### Output:

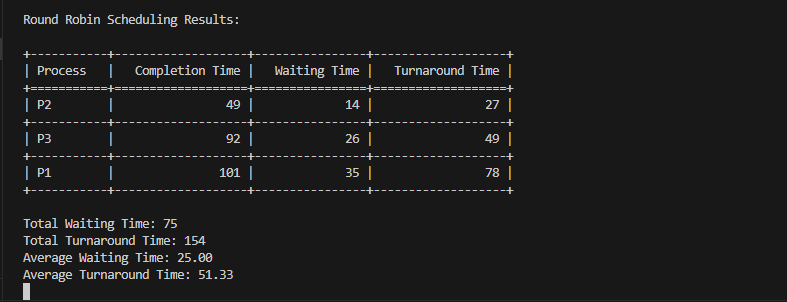


Figure 15:ROUND ROBIN.PY

### ‘

### Gantt chart:

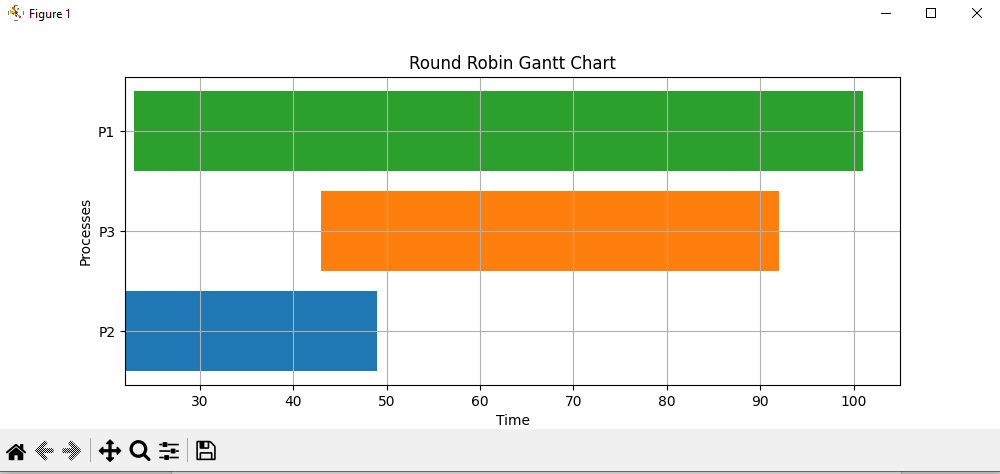


Figure 16:ROUND ROBIN.PY Gantt chart

# **6.4 Memory Management**

### 6.4.1: First fit.py

def \_\_init\_\_(self, memory\_blocks):

self.original\_blocks = memory\_blocks[:]

self.memory\_blocks = memory\_blocks[:]

self.allocation = {}

* self.memory\_blocks: This will change as memory gets allocated.
* self.allocation: A dictionary to record which process is allocated to which block.

def allocate(self, process\_name, process\_size):

for i, block\_size in enumerate(self.memory\_blocks):

if block\_size >= process\_size:

self.allocation[process\_name] = (i, process\_size)

self.memory\_blocks[i] -= process\_size

return True

return False

* This method implements the **First Fit** strategy.
* It iterates over each memory block:
  + If a block is large enough, the process is assigned to it.
  + The block’s size is reduced accordingly (self.memory\_blocks[i] -= process\_size).
* Returns True if successful; otherwise, False.

def get\_allocation(self):

return self.allocation

def get\_memory\_blocks(self):

return self.memory\_blocks

def get\_original\_blocks(self):

return self.original\_blocks

These are **getter methods** to inspect the state of the system:

* + get\_allocation() → Shows which process went into which block.
  + get\_memory\_blocks() → Current sizes of memory blocks after allocations.
  + get\_original\_blocks() → Unchanged, original memory block sizes for comparison.

### Output:

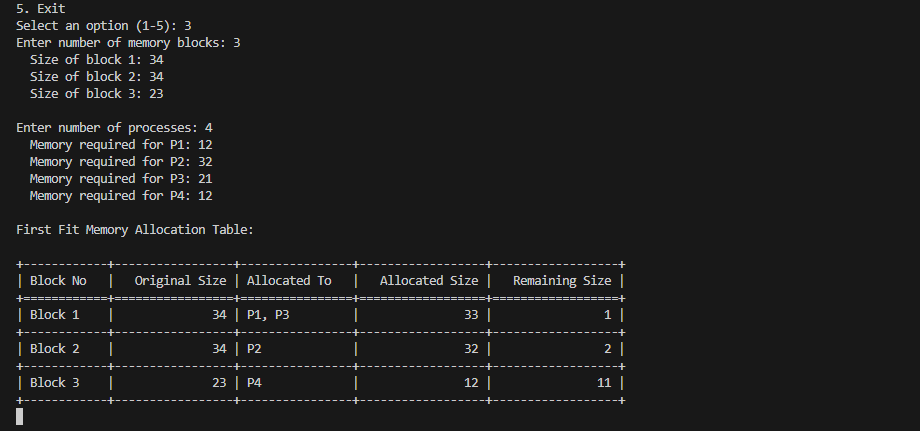


Figure 17: First fit.py

### Graph:

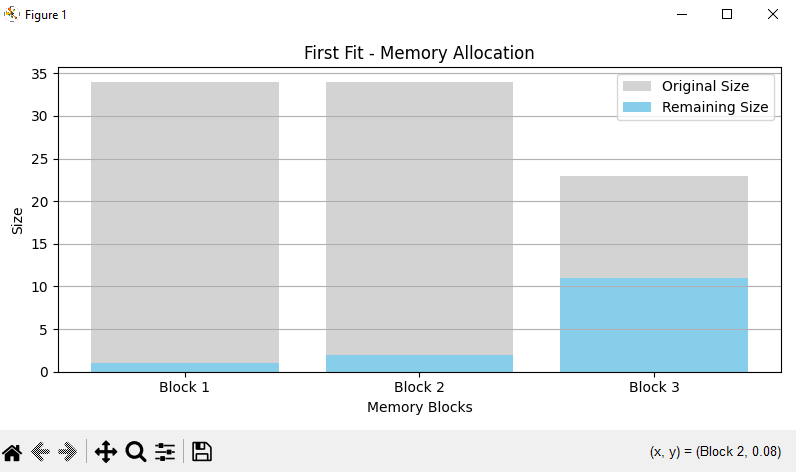


Figure 18:First fit.py Graph

### 6.4.2: Best fit.py

def \_\_init\_\_(self, memory\_blocks):

self.original\_blocks = memory\_blocks[:]

self.memory\_blocks = memory\_blocks[:]

self.allocation = {}

* This is the **constructor** used to set up the memory allocator.
* self.original\_blocks: Keeps the original block sizes (unchanged).
* self.memory\_blocks: A working copy that will shrink as memory is allocated.
* self.allocation: A dictionary to track which process is placed in which block.

def allocate(self, process\_name, process\_size):

best\_block\_index = -1

min\_waste = float('inf')

for i, block\_size in enumerate(self.memory\_blocks):

if block\_size >= process\_size:

waste = block\_size - process\_size

if waste < min\_waste:

min\_waste = waste

best\_block\_index = i

if best\_block\_index != -1:

self.allocation[process\_name] = (best\_block\_index, process\_size)

self.memory\_blocks[best\_block\_index] -= process\_size

return True

return False

* This method finds the **most efficient block** (least leftover space) for the process.
* min\_waste: Tracks the smallest unused space after fitting.
* best\_block\_index: Stores the index of the best-fitting block.
* Once the best fit is found, memory is allocated and block size is reduced.
* If no suitable block is found, returns False.

def get\_allocation(self):

return self.allocation

def get\_memory\_blocks(self):

return self.memory\_blocks

def get\_original\_blocks(self):

return self.original\_blocks

* These methods help **inspect memory state**:
  + get\_allocation(): Shows where each process was placed.
  + get\_memory\_blocks(): Displays the current state of memory blocks after allocation.
  + get\_original\_blocks(): Returns the initial state of memory (unchanged for reference).

### Output:

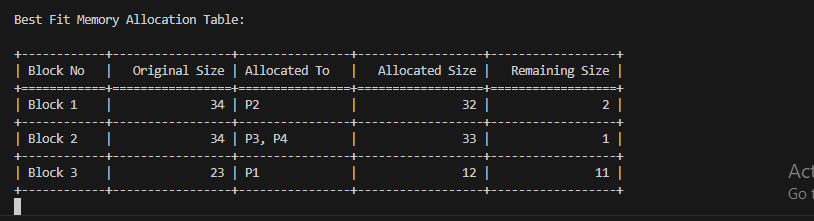


Figure 19:Best fit.py

### Graph:

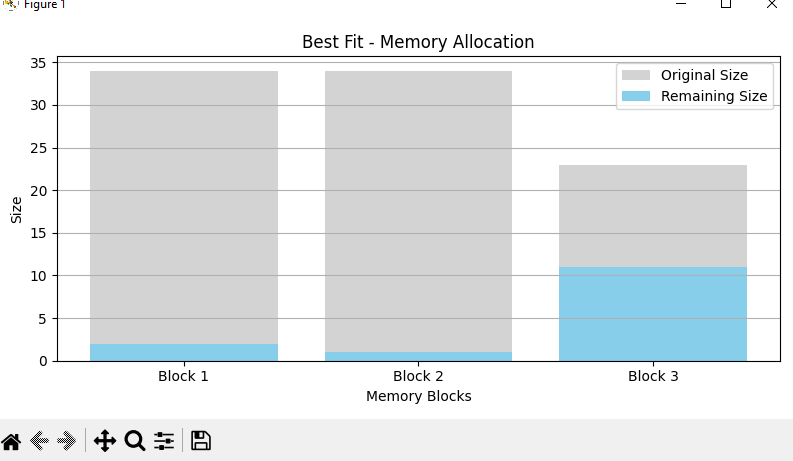


Figure 20: Best fit.py Graph

# **6.5 Deadlock**

### 6.5.1: Detection.py

def \_\_init\_\_(self, processes, total\_resources, allocation, request):

self.processes = processes

self.total\_resources = total\_resources

self.allocation = allocation

self.request = request

# ✅ Validate input dimensions before proceeding

if not all(len(row) == len(total\_resources) for row in allocation + request):

raise ValueError("❌ Mismatch: Each allocation and request row must match the number of resource types.")

# Calculate available resources

self.available = [

self.total\_resources[i] - sum(self.allocation[j][i] for j in range(len(processes)))

for i in range(len(total\_resources))

]:

* + processes: List of process names.
  + total\_resources: Total units of each resource.
  + allocation: How much each process currently holds.
  + request: Additional resources each process needs.
* Input validation ensures the number of resource types matches across rows.
* available: Calculates the remaining resources in the system using:

python

CopyEdit

available[i] = total[i] - sum(allocation[j][i] for all j)

def is\_safe(self):

work = self.available[:]

finish = [False] \* len(self.processes)

safe\_sequence = []

while True:

found = False

for i in range(len(self.processes)):

if not finish[i] and all(self.request[i][j] <= work[j] for j in range(len(work))):

for j in range(len(work)):

work[j] += self.allocation[i][j]

finish[i] = True

safe\_sequence.append(self.processes[i])

found = True

if not found:

break

if all(finish):

return True, safe\_sequence

else:

return False, [self.processes[i] for i, done in enumerate(finish) if not done]

* Implements the **Banker’s Algorithm** logic to test for a **safe state**.
* work: A copy of available resources to simulate allocation.
* finish: Tracks whether each process can finish with current resources.
* Tries to find a process that can complete (request ≤ work):
  + If yes: add its allocated resources back to work, mark as finished.
* If all processes finish, the system is in a **safe state**.
* If not, returns the list of processes that **might be deadlocked**.

def detect\_deadlock(self):

is\_safe\_state, info = self.is\_safe()

if is\_safe\_state:

return f"✅ No deadlock detected. Safe sequence: {', '.join(info)}"

else:

return f"⚠️ Deadlock detected! Processes involved: {', '.join(info)}"

* Calls the is\_safe() method.
* If system is safe: reports the **safe execution sequence**.
* If not: reports a **deadlock** and lists the **blocked processes**.
* Provides user-friendly status messages (with ✅ and ⚠️ for clarity).

### Output:

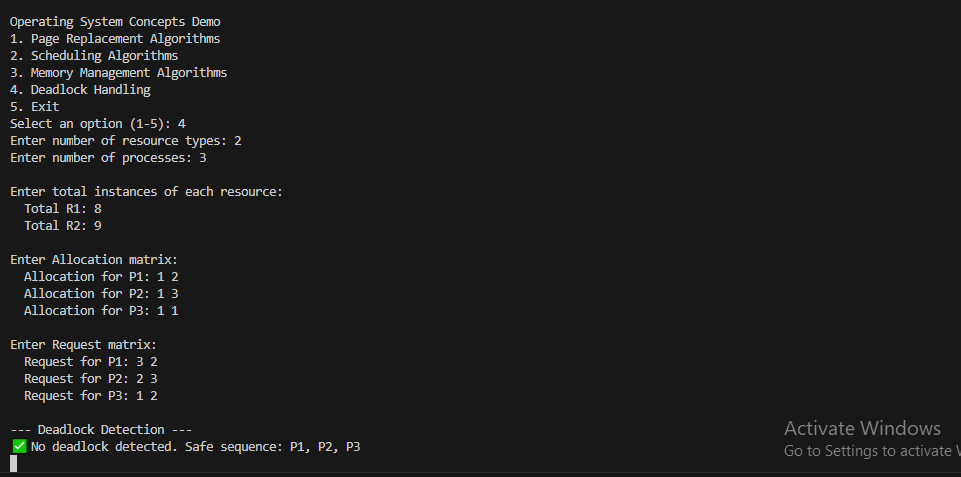


Figure 21:Detection.py

### Graph:

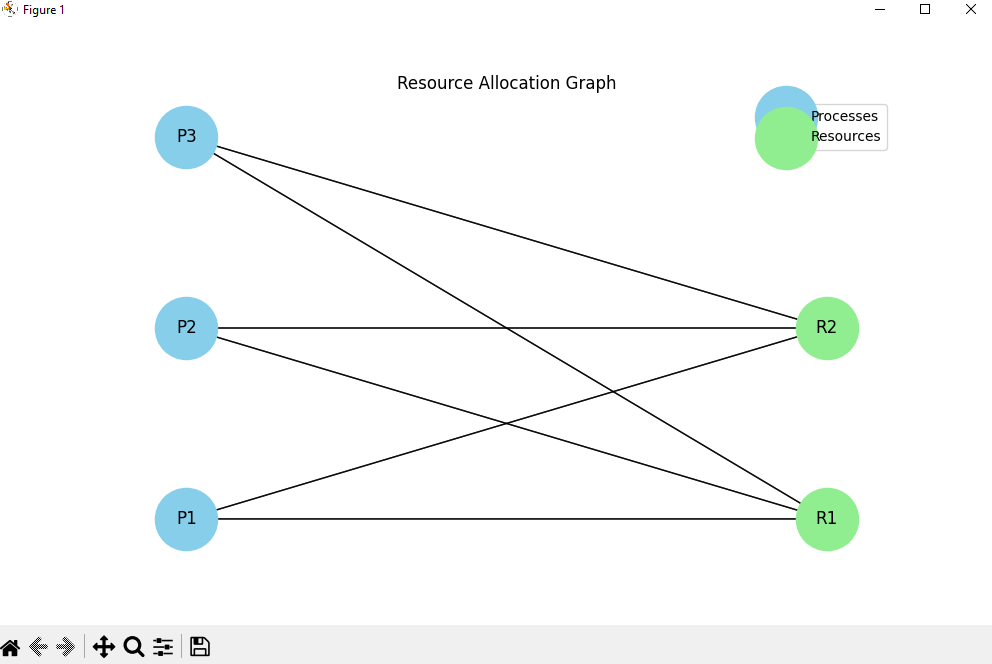


Figure 22:Detection.py Graph

# **7. Future Improvements / Conclusion**

**Future Improvements:**

* Add support for **multi-level feedback queue** and **priority-based scheduling**.
* Enable **user-defined scripting** with syntax validation.
* Introduce **user authentication and progress tracking**.
* Add **interactive quizzes** or assessments for learners.

**Conclusion:**

The OS Algorithm Visualizer successfully simulates essential operating system concepts in an interactive format. It simplifies complex processes using clear visuals and step-by-step execution. This project serves as an effective **educational tool** for students, teachers, and professionals seeking to understand core OS algorithms.