**OPERATING SYSTEMS PROJECT REPORT**

on

**IMPLEMENTATION OF SCHEDULING ALGORITHMS**

by

Sathvika

Name – 2

Name – 2

Table of Contents:

1. Introduction ………………………………………………………………………. 3
2. Basic Requirements ………………………………………………………………. 3
3. Advanced Requirements …………………………………………………………...17
4. New Algorithm………………………………………………………………...17
5. Dynamic Inputs ……………………………………………………………….20
6. Graphic User Interface (GUI) ………………………………………………….29
7. Analysis ………………………………………………………………………..41
8. Conclusion …………………………………………………………………………48
9. References …………………………………………………………………………49
10. **Introduction**

This project is centered on implementing key CPU scheduling algorithms in Python, including First-Come-First Serve (FCFS), Round Robin (RR), Shortest Process Next (SPN), Shortest Remaining Time (SRT), Highest Response Ratio Next (HRRN), and Multilevel Feedback Queue (MFQ). Designed to be interactive and user-friendly, it utilizes Python's capabilities to handle complex scheduling logic and Streamlit for creating an accessible graphical user interface (GUI). This combination allows for a dynamic exploration of algorithm behaviors, where users can input various process loads and immediately visualize the outcomes. In addition, a new algorithm has also been developed which is named as Adaptive Priority Scheduling Algorithm (APSA). A detailed analysis is also presented at the later section of report which highlights about the average waiting time and average turnaround time of various input types by various algorithms implemented as a part of basic and advance requirements.

1. **Basic Requirements:**

All CPU scheduling algorithms has been implemented in different files and placed in algorithms folder where they can be imported with a single import statement `from algorithms import fcfs as fcfs`. Each algorithm is designed to handle process scheduling in distinct ways, catering to different requirements and scenarios in an operating system.

**Static Inputs**

For testing, a set of static inputs is used: Arrival Times - [0, 1, 3, 4, 7] and Service Times - [10, 2, 5, 9, 7]. These inputs are consistently applied across all algorithms to evaluate and compare their performance.

**Print Results Feature**

Each algorithm includes a **print\_results** boolean parameter. When set to **True**, the algorithms output detailed information, including the waiting and turnaround times for each process.

Below are the implementation details of various algorithms implemented as part of the static requirements.

**a) First Come First Serve(FCFS) :** FCFS is the simplest scheduling method, operating on a first-come, first-serve basis. Processes are executed in the order they arrive, without preemption. It's straightforward but can lead to long waiting times for processes arriving later. The function signature is **fcfs(arrival\_times, service\_times, print\_results)**. It outputs the process execution sequence based on arrival times. Below is the code implementation of the algorithm.

"""

First-Come-First-Served Scheduling Algorithm

"""

\_\_all\_\_ = ["fcfs"]

def fcfs(arrival\_times, service\_times, print\_results=False):

"""

First-Come-First-Served Scheduling Algorithm

This function implements the First-Come-First-Served (FCFS) scheduling algorithm.

It takes a list of arrival times and service times of processes as input and returns

the waiting times and turnaround times of each process.

Parameters:

arrival\_times (list): List of arrival times of processes.

service\_times (list): List of service (burst) times of processes.

print\_results (bool): If True, prints the scheduling details.

Returns:

waiting\_times (list): List of waiting times of each process.

turnaround\_times (list): List of turnaround times of each process.

"""

n = len(arrival\_times)

waiting\_times = [0] \* n

turnaround\_times = [0] \* n

start\_time = 0

for i in range(n):

if start\_time < arrival\_times[i]:

start\_time = arrival\_times[i]

waiting\_times[i] = start\_time - arrival\_times[i]

start\_time += service\_times[i]

turnaround\_times[i] = waiting\_times[i] + service\_times[i]

if print\_results:

print("First-Come-First-Served Scheduling")

print("Process\tArrival\tService\tWaiting\tTurnaround")

for i in range(n):

print(

f"{i+1}\t{arrival\_times[i]}\t{service\_times[i]}\t{waiting\_times[i]}\t{turnaround\_times[i]}"

)

print(f"Average Waiting Time: {sum(waiting\_times)/n}")

print(f"Average Turnaround Time: {sum(turnaround\_times)/n}")

return waiting\_times, turnaround\_times

A screen shot of a black and white screen

Description automatically generated

**Figure 2.1 FCFS**

**b) Round Robin (RR):** RR assigns a fixed time slice to each process in a cyclic order. If a process doesn't complete within its time slice, it's moved to the end of the queue. This ensures a more equitable CPU allocation. The signature is **round\_robin(arrival\_times, service\_times, time\_quantum, print\_results)**. It's designed to ensure all processes get an equal share of the CPU.

Time quantum = 7

Below is the implementation code for Round Robin Algorithm

"""

Round Robin Scheduling Algorithm

"""

\_\_all\_\_ = ["round\_robin"]

def round\_robin(arrival\_times, service\_times, quantum, print\_results=False):

"""

Round Robin Scheduling Algorithm

Parameters:

arrival\_times (list): List of arrival times of processes.

service\_times (list): List of service (burst) times of processes.

quantum (int): Time quantum for the round-robin scheduling.

print\_results (bool): If True, prints the scheduling details.

Returns:

waiting\_times (list): List of waiting times for each process.

turnaround\_times (list): List of turnaround times for each process.

"""

n = len(arrival\_times)

remaining\_times = list(service\_times)

waiting\_times = [0] \* n

turnaround\_times = [0] \* n

t = 0 # Current time

while True:

done = True

for i in range(n):

if remaining\_times[i] > 0:

done = False

if remaining\_times[i] > quantum:

t += quantum

remaining\_times[i] -= quantum

else:

t += remaining\_times[i]

waiting\_times[i] = t - service\_times[i] - arrival\_times[i]

remaining\_times[i] = 0

if done:

break

for i in range(n):

turnaround\_times[i] = service\_times[i] + waiting\_times[i]

if print\_results:

print("Round Robin Scheduling")

print(f"Time Quantum: {quantum}")

print("Process\tArrival\tService\tWaiting\tTurnaround")

for i in range(n):

print(

f"{i+1}\t{arrival\_times[i]}\t{service\_times[i]}\t{waiting\_times[i]}\t{turnaround\_times[i]}"

)

print(f"Average Waiting Time: {sum(waiting\_times)/n}")

print(f"Average Turnaround Time: {sum(turnaround\_times)/n}")

return waiting\_times, turnaround\_times

Execution Output:

A screenshot of a computer screen

Description automatically generated

**Figure 2.2 RR**

**c) Shortest Process Next (SPN):** SPN selects the process with the shortest estimated run time for execution next. It's non-preemptive, favoring shorter jobs and reducing overall average waiting time but can cause longer jobs to wait excessively. The function is **spn(arrival\_times, service\_times, print\_results)**, which favors shorter jobs to reduce the average waiting time.

def spn(arrival\_times, service\_times, print\_results=False):

"""

Shortest Process Next (SPN) Scheduling Algorithm

This function implements the Shortest Process Next (SPN) scheduling algorithm.

It takes a list of arrival times and service times as input and returns the waiting times and turnaround times for each process.

Parameters:

- arrival\_times: List of arrival times for each process.

- service\_times: List of service (burst) times for each process.

- print\_results: Boolean value indicating whether to print the process details and summary. Default is False.

Returns:

- waiting\_times: List of waiting times for each process.

- turnaround\_times: List of turnaround times for each process.

"""

n = len(arrival\_times)

waiting\_times = [0] \* n

finish\_times = [0] \* n

service\_times\_remaining = list(service\_times)

time = 0

processes = set(range(n))

while processes:

available\_processes = [i for i in processes if arrival\_times[i] <= time]

if not available\_processes:

time += 1

continue

shortest\_process = min(

available\_processes, key=lambda i: service\_times\_remaining[i]

)

processes.remove(shortest\_process)

waiting\_times[shortest\_process] = time - arrival\_times[shortest\_process]

time += service\_times\_remaining[shortest\_process]

finish\_times[shortest\_process] = time

turnaround\_times = [finish\_times[i] - arrival\_times[i] for i in range(n)]

if print\_results:

print("Shortest Process Next Scheduling")

print("Process\tArrival\tService\tWaiting\tTurnaround")

for i in range(n):

print(

f"{i + 1}\t{arrival\_times[i]}\t{service\_times[i]}\t{waiting\_times[i]}\t{turnaround\_times[i]}"

)

print(f"\nAverage Waiting Time: {sum(waiting\_times) / n:.2f}")

print(f"Average Turnaround Time: {sum(turnaround\_times) / n:.2f}")

return waiting\_times, turnaround\_times

**A screen shot of a black screen

Description automatically generated**

**Figure 2.3 SPN**

**d) Shortest Remaining Time (SRT):** SRT is like SPN but preemptive. It selects the process with the shortest remaining time to completion. This approach often results in lower average waiting times, especially for short jobs. The signature is **srt(arrival\_times, service\_times, print\_results)**, ideally suited for quickly executing short jobs.

def srt(arrival\_times, service\_times, print\_results=False):

"""

Shortest Remaining Time (SRT) Scheduling Algorithm

Parameters:

- arrival\_times: List of arrival times

- service\_times: List of service (burst) times

- print\_results: Boolean, if True, print the process details in table format

Returns:

- waiting\_times: List of waiting times for each process

- turnaround\_times: List of turnaround times for each process

"""

n = len(arrival\_times)

waiting\_times = [0] \* n

finish\_times = [0] \* n

remaining\_times = list(service\_times)

time = 0

completed = 0

while completed != n:

# Find process with minimum remaining time

shortest = min(

[

i

for i in range(n)

if arrival\_times[i] <= time and remaining\_times[i] > 0

],

key=lambda x: remaining\_times[x],

default=-1,

)

if shortest == -1:

time += 1

continue

remaining\_times[shortest] -= 1

time += 1

if remaining\_times[shortest] == 0:

completed += 1

finish\_times[shortest] = time

waiting\_times[shortest] = (

finish\_times[shortest]

- arrival\_times[shortest]

- service\_times[shortest]

)

turnaround\_times = [finish\_times[i] - arrival\_times[i] for i in range(n)]

if print\_results:

print("\nShortest Remaining Time (SRT) Scheduling Algorithm")

print("Process\tArrival Time\tService Time\tWaiting Time\tTurnaround Time")

for i in range(n):

print(

f"{i + 1}\t\t{arrival\_times[i]}\t\t{service\_times[i]}\t\t{waiting\_times[i]}\t\t{turnaround\_times[i]}"

)

print(f"\nAverage Waiting Time: {sum(waiting\_times) / n:.2f}")

print(f"Average Turnaround Time: {sum(turnaround\_times) / n:.2f}")

return waiting\_times, turnaround\_times

A screen shot of a computer

Description automatically generated

**Figure 2.4 SRT**

**e) Highest Response Ratio Next (HRRN):** HRRN calculates a response ratio for each process and schedules the one with the highest ratio next. This ratio is based on waiting time and service time, balancing the needs of both short and long processes. The function **hrrn(arrival\_times, service\_times, print\_results)** balances between shorter and longer processes, aiming for optimal turnaround time.

def hrrn(arrival\_times, service\_times, print\_results=False):

"""

Highest Response Ratio Next (HRRN) Scheduling Algorithm

This function implements the Highest Response Ratio Next (HRRN) scheduling algorithm.

It schedules the processes based on their response ratio, which is calculated as the ratio

of the sum of the waiting time and the service time to the service time.

Parameters:

- arrival\_times: List of arrival times for each process.

- service\_times: List of service (burst) times for each process.

- print\_results: Boolean value indicating whether to print the process details in table format.

Returns:

- waiting\_times: List of waiting times for each process.

- turnaround\_times: List of turnaround times for each process.

"""

n = len(arrival\_times)

waiting\_times = [0] \* n

finish\_times = [0] \* n

start\_times = [0] \* n

remaining\_times = list(service\_times)

time = 0

processes = set(range(n))

while processes:

available\_processes = [

(i, ((time - arrival\_times[i]) + service\_times[i]) / service\_times[i])

for i in processes

if arrival\_times[i] <= time

]

if not available\_processes:

time += 1

continue

# Process with the highest response ratio

next\_process = max(available\_processes, key=lambda x: x[1])[0]

processes.remove(next\_process)

waiting\_times[next\_process] = time - arrival\_times[next\_process]

time += service\_times[next\_process]

finish\_times[next\_process] = time

start\_times[next\_process] = (

waiting\_times[next\_process] + arrival\_times[next\_process]

)

turnaround\_times = [service\_times[i] + waiting\_times[i] for i in range(n)]

if print\_results:

print("Highest Response Ratio Next (HRRN) Scheduling Algorithm")

print(

"Process\tArrival Time\tService Time\tWaiting Time\tStart Time\tTurnaround Time"

)

for i in range(n):

print(

f"{i + 1}\t\t{arrival\_times[i]}\t\t{service\_times[i]}\t\t{waiting\_times[i]}\t\t{start\_times[i]}\t\t{turnaround\_times[i]}"

)

print(f"\nAverage Waiting Time: {sum(waiting\_times) / n:.2f}")

print(f"Average Turnaround Time: {sum(turnaround\_times) / n:.2f}")

return waiting\_times, turnaround\_times

A screen shot of a black screen

Description automatically generated

**Figure 2.5 HRRN**

**f) Multilevel Feedback Queue (MFQ):** MFQ uses multiple queues with different scheduling criteria. Processes move between queues based on their age or execution characteristics. It's designed to dynamically prioritize tasks, offering flexibility and responsiveness. The signature is **mfq(arrival\_times, service\_times, t1, t2, print\_results)**. It dynamically adjusts a process's priority based on its behavior and requirements.

Below are the important implementation details of Multilevel Feedback Queue:

It has three queues:

1. Queue 1 (highest priority): Round Robin with t1 time quanta.
2. Queue 2: Round Robin with t2 time quanta.
3. Queue 3 (lowest priority): First-Come-First-Serve (FCFS).

The algorithm works as follows:

* Processes are initially placed in the highest priority queue (Queue 1).
* If a process uses up all its time quanta in Queue 1, it's moved to Queue 2.
* Similarly, if it uses up all its time in Queue 2, it's moved to Queue 3.
* In Queue 3, processes are executed on a FCFS basis without time quantum.
* If a new process arrives in a higher priority queue, it preempts the currently running process in a lower priority queue.

class Process:

def \_\_init\_\_(self, pid, arrival\_time, burst\_time):

self.pid = pid

self.arrival\_time = arrival\_time

self.burst\_time = burst\_time

self.remaining\_time = burst\_time

self.waiting\_time = 0

self.completed = False

self.priority = None

def get\_process\_by\_pid(processes, pid):

for process in processes:

if process.pid == pid:

return process

return None

def mlfq(arrival\_times, service\_times, t1, t2, print\_results=False):

"""

Implements the Multi-Level Feedback Queue (MLFQ) scheduling algorithm.

Args:

arrival\_times (list): List of arrival times for each process.

service\_times (list): List of service times for each process.

t1 (int): Time quantum for the first queue.

t2 (int): Time quantum for the second queue.

print\_results (bool, optional): Whether to print the scheduling results. Defaults to False.

Returns:

tuple: A tuple containing the waiting times and turnaround times for each process.

"""

n = len(arrival\_times)

processes = [Process(i, arrival\_times[i], service\_times[i]) for i in range(n)]

queues = [Queue() for \_ in range(3)]

time\_quantum = [t1, t2, float("inf")]

current\_time = 0

current\_process = None

gantt\_chart = []

while any(p.remaining\_time > 0 for p in processes):

for process in processes:

if (

process.arrival\_time <= current\_time

and not process.completed

and process.priority is None

):

queues[0].put(process.pid)

process.priority = 0

for i in range(len(queues)):

if not queues[i].empty() and (

current\_process is None or i < current\_process.priority

):

if current\_process:

queues[current\_process.priority].put(current\_process.pid)

pid = queues[i].get()

current\_process = get\_process\_by\_pid(processes, pid)

break

if current\_process:

current\_process.remaining\_time -= 1

time\_quantum[current\_process.priority] -= 1

if current\_process.remaining\_time == 0:

current\_process.waiting\_time = (

current\_time

- current\_process.arrival\_time

- current\_process.burst\_time

+ 1

)

current\_process.completed = True

current\_process = None

elif time\_quantum[current\_process.priority] == 0:

if current\_process.priority < 2:

queues[current\_process.priority + 1].put(current\_process.pid)

current\_process.priority += 1

current\_process = None

time\_quantum = [8, 16, float("inf")]

current\_time += 1

if print\_results:

print("Multi-Level Feedback Queue Scheduling")

print("First Queue: Time Quantum = 8")

print("Second Queue: Time Quantum = 16")

print("Third Queue: FCFS")

print("Process\tArrival Time\tService Time\tWaiting Time\tTurnaround Time")

for i in range(n):

print(

f"{i + 1}\t\t{arrival\_times[i]}\t\t{service\_times[i]}\t\t{processes[i].waiting\_time}\t\t{processes[i].waiting\_time + service\_times[i]}"

)

waiting\_times = [p.waiting\_time for p in processes]

turnaround\_times = [p.waiting\_time + p.burst\_time for p in processes]

print(f"\nAverage Waiting Time: {sum(waiting\_times) / n:.2f}")

print(f"Average Turnaround Time: {sum(turnaround\_times) / n:.2f}")

return waiting\_times, turnaround\_times

**A screenshot of a computer screen

Description automatically generated**

**Figure 2.6 Multi Level Feedback Queue**

Below is the code to run all the algorithms in a single click using the static inputs.

"""

This file contains static inputs for testing the algorithms which covers the

basic requirements of the project.

"""

from algorithms.fcfs import fcfs

from algorithms.hrrn import hrrn

from algorithms.srt import srt

from algorithms.spn import spn

from algorithms.rr import round\_robin

from algorithms.mfq import mlfq as mfq

from algorithms.custom import apsa

# define static inputs

arrival\_times = [0, 1, 3, 4, 7]

service\_times = [10, 2, 5, 9, 7]

# time quanta for some algorithms

time\_quanta = 7

# now call all of the algorithms with print\_results=True

fcfs(arrival\_times, service\_times, print\_results=True)

print("\n")

round\_robin(arrival\_times, service\_times, time\_quanta, print\_results=True)

print("\n")

spn(arrival\_times, service\_times, print\_results=True)

print("\n")

srt(arrival\_times, service\_times, print\_results=True)

print("\n")

hrrn(arrival\_times, service\_times, print\_results=True)

print("\n")

mfq(arrival\_times, service\_times, 8, 16, print\_results=True)

print("\n")

apsa(arrival\_times, service\_times, 0.5, 10, print\_results=True)

1. **Advanced Requirements:**
2. **New Algorithm:**

Designing a new scheduling algorithm can be quite a challenge, but it's also an opportunity to be creative and innovate. For this task, I have come up with a new algorithm that combines aspects of existing algorithms to cater different types of processes. Below are the algorithm details along with the code implementation.

The Adaptive Priority Scheduling Algorithm is a preemptive scheduler that dynamically adjusts the priority of processes based on their burst time, arrival time, and the amount of time they have already been waiting. It aims to minimize the average waiting time and turnaround time, especially for I/O-bound and short tasks, while also ensuring that CPU-bound tasks are processed efficiently.

**Concept**:

* Each process is assigned an initial priority based on its estimated burst time and arrival time.
* The priority is recalculated at regular intervals and after every process execution.
* Priorities are adjusted to ensure that shorter and I/O-bound processes are given preference, while also preventing starvation of longer, CPU-bound processes.

**Initial Priority Calculation**:

* Shorter processes and processes that have just arrived are given higher priority.
* The formula for priority is : **Priority = 1 / (Burst Time + Arrival Time Factor)**
* The Arrival Time Factor ensures that newly arrived processes get a boost in priority.

**Dynamic Priority Adjustment**:

* After each execution cycle, the priority of the waiting processes is recalculated.
* If a process has been waiting, its priority is increased, which can be determined by the formula: **Priority += Waiting Time Factor**
* If a process exhausts its time slice without completing, its priority is slightly reduced to give way to other processes.

**Execution**:

* At each decision point, the scheduler selects the process with the highest priority.
* If a new process arrives with higher priority than the currently running process, a context switch occurs.

**Starvation Prevention**:

* If a process's waiting time exceeds a certain threshold, its priority is boosted significantly to ensure it gets CPU time.
* Every process is guaranteed to get a minimum amount of CPU time after a certain waiting period.

Below is the python implementation of the above algorithm

def apsa(

arrival\_times,

burst\_times,

WAITING\_TIME\_FACTOR,

ARRIVAL\_TIME\_FACTOR,

print\_results=False,

):

"""

Implements the Adaptive Priority Scheduling Algorithm (APSA).

Args:

arrival\_times (list): List of arrival times for each process.

burst\_times (list): List of burst times for each process.

WAITING\_TIME\_FACTOR (float): Factor to determine the waiting time boost for processes.

ARRIVAL\_TIME\_FACTOR (float): Factor to determine the arrival time boost for processes.

print\_results (bool, optional): Flag to print the results. Defaults to False.

Returns:

tuple: A tuple containing the waiting times and turnaround times for each process.

"""

num\_processes = len(arrival\_times)

waiting\_times = [0] \* num\_processes

turnaround\_times = [0] \* num\_processes

priorities = [1 / (burst + arrival\_times[i]) for i, burst in enumerate(burst\_times)]

completed = [False] \* num\_processes

time = 0

queue = []

while not all(completed):

# Add processes to the queue based on arrival time

queue.extend(

[

i

for i, arrival in enumerate(arrival\_times)

if arrival <= time and not completed[i] and i not in queue

]

)

# Recalculate priorities for waiting processes

for i in queue:

if not completed[i]:

waiting\_times[i] = time - arrival\_times[i]

# Boost priority for processes that have been waiting too long

waiting\_boost = max(waiting\_times[i] \* WAITING\_TIME\_FACTOR, 1)

priorities[i] = (

1 / (burst\_times[i] + arrival\_times[i] / ARRIVAL\_TIME\_FACTOR)

+ waiting\_boost

)

# Select process with highest priority

if queue:

current\_process = max(queue, key=lambda i: priorities[i])

queue.remove(current\_process)

# Process execution simulation for one time unit

burst\_times[current\_process] -= 1

time += 1

# Check if the process is completed

if burst\_times[current\_process] <= 0:

completed[current\_process] = True

turnaround\_times[current\_process] = (

time - arrival\_times[current\_process]

)

else:

time += 1

if print\_results:

print("Adaptive Priority Scheduling Algorithm (APSA):")

print("WAITING\_TIME\_FACTOR:", WAITING\_TIME\_FACTOR)

print("ARRIVAL\_TIME\_FACTOR:", ARRIVAL\_TIME\_FACTOR)

print("Process\tArrival\tBurst\tWaiting\tTurnaround")

for i in range(num\_processes):

print(

f"{i+1}\t{arrival\_times[i]}\t{burst\_times[i]}\t{waiting\_times[i]}\t{turnaround\_times[i]}"

)

print("Average Waiting Time:", sum(waiting\_times) / num\_processes)

print("Average Turnaround Time:", sum(turnaround\_times) / num\_processes)

return waiting\_times, turnaround\_times

1. **Dynamic Inputs:**

The implementation was enhanced with the capability to accept dynamic inputs, allowing real-time simulation and testing of the algorithms with varying process loads. For every input taken, assertions are made to make sure they are given in right format.

Below is the code that presents the user with all the available algorithms and then takes the inputs from the user interactively.

from algorithms.fcfs import fcfs

from algorithms.hrrn import hrrn

from algorithms.srt import srt

from algorithms.spn import spn

from algorithms.rr import round\_robin

from algorithms.mfq import mlfq as mfq

from algorithms.custom import apsa

def get\_positive\_int(prompt):

while True:

try:

value = int(input(prompt).strip())

if value <= 0:

raise ValueError("The number must be positive.")

return value

except ValueError as ve:

print(f"Invalid input: {ve}")

def get\_positive\_float(prompt):

while True:

try:

value = float(input(prompt).strip())

if value <= 0:

raise ValueError("The number must be positive.")

return value

except ValueError as ve:

print(f"Invalid input: {ve}")

def get\_input(prompt, count):

while True:

try:

values = list(map(int, input(prompt).strip().split()))

if len(values) != count:

raise ValueError(f"Exactly {count} numbers are required.")

if any(v < 0 for v in values):

raise ValueError("Negative numbers are not allowed.")

return values

except ValueError as ve:

print(f"Invalid input: {ve}")

def run\_algorithm(algo, name, num\_processes, is\_rr=False, is\_mfq=False):

arrival\_times = get\_input(

"Enter the processes' arrival times separated by space: ", num\_processes

)

service\_times = get\_input(

"Enter the processes' service times separated by space: ", num\_processes

)

if is\_rr:

time\_quantum = get\_positive\_int("Enter time quantum for Round Robin: ")

algo(arrival\_times, service\_times, time\_quantum, print\_results=True)

elif is\_mfq:

t1 = get\_positive\_int("Enter time quantum t1 for MLFQ: ")

t2 = get\_positive\_int("Enter time quantum t2 for MLFQ: ")

algo(arrival\_times, service\_times, t1, t2, print\_results=True)

elif "APSA" in name:

waiting\_time\_factor = get\_positive\_float(

"Enter the waiting time factor for APSA: "

)

arrival\_time\_factor = get\_positive\_int(

"Enter the arrival time factor for APSA: "

)

algo(

arrival\_times,

service\_times,

waiting\_time\_factor,

arrival\_time\_factor,

print\_results=True,

)

else:

algo(arrival\_times, service\_times, print\_results=True)

def select\_algorithm():

algorithms = {

"1": ("First-Come-First-Served (FCFS)", fcfs, False, False),

"2": ("Round Robin (RR)", round\_robin, True, False),

"3": ("Shortest Process Next (SPN)", spn, False, False),

"4": ("Shortest Remaining Time (SRT)", srt, False, False),

"5": ("Highest Response Ratio Next (HRRN)", hrrn, False, False),

"6": ("Multilevel Feedback Queue (MLFQ)", mfq, False, True),

"7": ("Adaptive Priority Scheduling Algorithm (APSA)", apsa, False, False),

}

while True:

print("\nSelect the scheduling algorithm to run:")

for key, (name, \_, \_, \_) in algorithms.items():

print(f"{key}. {name}")

print("0. Exit")

choice = input("Enter your choice: ").strip()

if choice == "0":

break

if choice in algorithms:

name, algo, is\_rr, is\_mfq = algorithms[choice]

print(f"\n{name}:")

num\_processes = get\_positive\_int("Enter the number of processes: ")

run\_algorithm(algo, name, num\_processes, is\_rr, is\_mfq)

else:

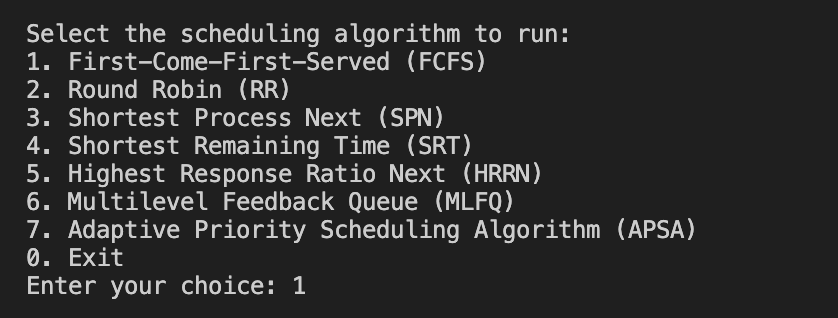
print("Invalid selection. Please try again.")

if \_\_name\_\_ == "\_\_main\_\_":

select\_algorithm()

Above dynamic input feature of this implementation enhances the project significantly. It allows real-time simulation and testing with various process loads. Each input is thoroughly checked to ensure it's in the correct format.

When the program runs, the user sees a prompt:



The user is then guided to select an algorithm. Following this, the number of processes is requested, along with the arrival and service times for each, inputted as space-separated values. For algorithms like Round Robin and MFQ, time quanta are also asked for. The program meticulously validates every input to ensure accuracy.

A robust input validation mechanism is in place: A screenshot of a computer program

Description automatically generated

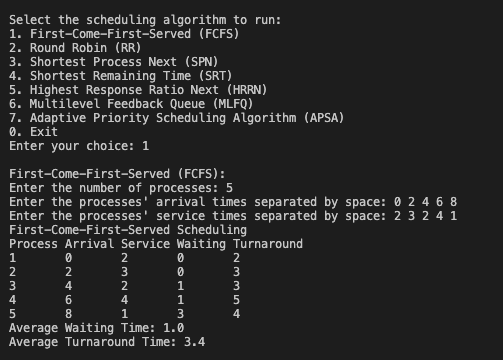
**Figure 3.1 Invalid Input Detection**

As shown above, if an invalid input is entered, it asks again until valid input is entered.

This ensures that the simulations run smoothly and accurately, reflecting the real-world scenarios they are designed to emulate.

Here's what the process looks like for each algorithm:

**FCFS:**

****

**Figure 3.2 FCFS Dynamic Output**

**RR:**

**A screenshot of a computer program

Description automatically generated**

**Figure 3.3 RR Dynamic Output**

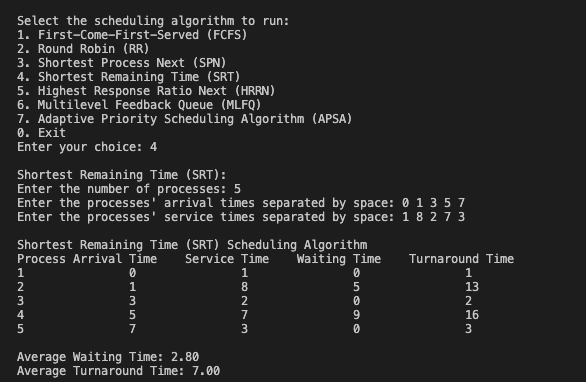
**SPN:**

**A screenshot of a computer program

Description automatically generated**

**Figure 3.4 SPN Dynamic Output**

**SRT:**

****

**Figure 3.5 SRT Dynamic Output**

**HRRN:**

**A screenshot of a computer program

Description automatically generated**

**Figure 3.6 HRRN Dynamic Output**

**MFS:**

**A screenshot of a computer program

Description automatically generated**

**Figure 3.7 MFS Dynamic Output**

**APSA:**

**A screenshot of a computer program

Description automatically generated**

**C) Graphical UI:** Utilizing Streamlit, an intuitive and interactive GUI was developed. This interface enables users to easily input process data, select algorithms, and view results, making the exploration of scheduling algorithms more accessible.

**A screenshot of a computer

Description automatically generated**

This Python code creates a web app simulating scheduling algorithms for processes.

1. User selects:
   * Algorithm (FCFS, Round Robin, etc.)
   * Number of processes
2. For each process:
   * User inputs arrival and service times
3. Conditional inputs:
   * Time quantum for Round Robin/MFQ
   * Waiting/arrival time factors for APSA
4. Click 'Run':
   * Algorithm runs with user-provided data
5. Output:
   * Table showing process details (arrival, service, waiting, turnaround times)
   * Average waiting and turnaround times

This interactive app leverages libraries like Streamlit and Pandas for a seamless user experience and data visualization. The functions (run\_algorithm and main) handle data processing, algorithm execution, and output generation.

GUI app can also be deployed to cloud in a single click.

**A screenshot of a computer program

Description automatically generated**

**FCFS**

**A screenshot of a computer

Description automatically generated**

**Round Robin**

**A screenshot of a computer

Description automatically generated**

**SPN**

**A screenshot of a program

Description automatically generated**

**SRT**

**A screenshot of a computer

Description automatically generated**

**HRNN**

**A screenshot of a computer

Description automatically generated**

**MFQ**

**A screenshot of a black screen

Description automatically generated**

**ASPA**

**Code for GUI is as below:**

import streamlit as st

import pandas as pd

from algorithms.fcfs import fcfs

from algorithms.hrrn import hrrn

from algorithms.srt import srt

from algorithms.spn import spn

from algorithms.rr import round\_robin

from algorithms.mfq import mlfq as mfq

from algorithms.custom import apsa

def run\_algorithm(

algo\_func,

arrival\_times,

service\_times,

time\_quantum=None,

waiting\_time\_factor=None,

arrival\_time\_factor=None,

time\_quantum2=None,

):

service\_times1 = service\_times.copy()

arrival\_times1 = arrival\_times.copy()

if time\_quantum and not time\_quantum2:

waiting\_times, turnaround\_times = algo\_func(

arrival\_times, service\_times, time\_quantum, print\_results=False

)

elif time\_quantum2:

waiting\_times, turnaround\_times = algo\_func(

arrival\_times,

service\_times,

time\_quantum,

time\_quantum2,

print\_results=False,

)

elif waiting\_time\_factor and arrival\_time\_factor:

waiting\_times, turnaround\_times = algo\_func(

arrival\_times,

service\_times,

waiting\_time\_factor,

arrival\_time\_factor,

print\_results=False,

)

else:

waiting\_times, turnaround\_times = algo\_func(

arrival\_times, service\_times, print\_results=False

)

# Constructing the table

process\_ids = list(range(1, len(arrival\_times) + 1))

data = {

"Process": process\_ids,

"Arrival Time": arrival\_times1,

"Service Time": service\_times1,

"Waiting Time": waiting\_times,

"Turnaround Time": turnaround\_times,

}

df = pd.DataFrame(data)

st.table(df)

# Calculate and display averages

avg\_waiting\_time = sum(waiting\_times) / len(waiting\_times)

avg\_turnaround\_time = sum(turnaround\_times) / len(turnaround\_times)

st.write(f"Average Waiting Time: {avg\_waiting\_time:.2f}")

st.write(f"Average Turnaround Time: {avg\_turnaround\_time:.2f}")

def main():

st.title("Scheduling Algorithm Simulator")

# Selection of the algorithm

algorithm = st.selectbox(

"Select the scheduling algorithm:",

("FCFS", "Round Robin", "SPN", "SRT", "HRRN", "MFQ", "APSA"),

)

# Input for processes

num\_processes = st.number\_input(

"Enter the number of processes:",

min\_value=1,

value=1,

step=1,

key="num\_processes",

)

# Use a form for batch input submission

form = st.form(key="processes\_form")

arrival\_times = []

service\_times = []

for i in range(num\_processes):

cols = form.columns(2)

with cols[0]:

arrival\_time = st.number\_input(

f"Process {i+1} - Arrival Time:", min\_value=0, key=f"arrival\_{i}"

)

with cols[1]:

service\_time = st.number\_input(

f"Process {i+1} - Service Time:", min\_value=0, key=f"service\_{i}"

)

arrival\_times.append(arrival\_time)

service\_times.append(service\_time)

# Conditional input for time quantum if Round Robin or MFQ is selected

time\_quantum = None

if algorithm in ["Round Robin", "MFQ"]:

time\_quantum = form.number\_input(

"Enter time quantum:", min\_value=1, value=1, key="time\_quantum"

)

time\_quantum2 = None

if algorithm == "MFQ":

time\_quantum2 = form.number\_input(

"Enter time quantum for second queue:",

min\_value=1,

value=1,

key="time\_quantum2",

)

waiting\_time\_factor = None

arrival\_time\_factor = None

if algorithm == "APSA":

waiting\_time\_factor = form.number\_input(

"Enter the waiting time factor for APSA: ",

min\_value=0.0,

value=0.5,

key="waiting\_time\_factor",

)

arrival\_time\_factor = form.number\_input(

"Enter the arrival time factor for APSA: ",

min\_value=0,

value=10,

key="arrival\_time\_factor",

)

submit\_button = form.form\_submit\_button(label="Run {}".format(algorithm))

if submit\_button:

algo\_mapping = {

"FCFS": fcfs,

"Round Robin": round\_robin,

"SPN": spn,

"SRT": srt,

"HRRN": hrrn,

"MFQ": mfq,

"APSA": apsa,

}

# Call the corresponding algorithm function

run\_algorithm(

algo\_mapping[algorithm],

arrival\_times,

service\_times,

time\_quantum,

waiting\_time\_factor,

arrival\_time\_factor,

time\_quantum2,

)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**D) Analysis:**

* The analysis involved running the six scheduling algorithms on various input sets. Each set represents a different workload type, providing insight into how each algorithm performs under diverse scenarios.
* The results are captured in a table format, listing the input set, the algorithm used, and the corresponding average turnaround and waiting times.

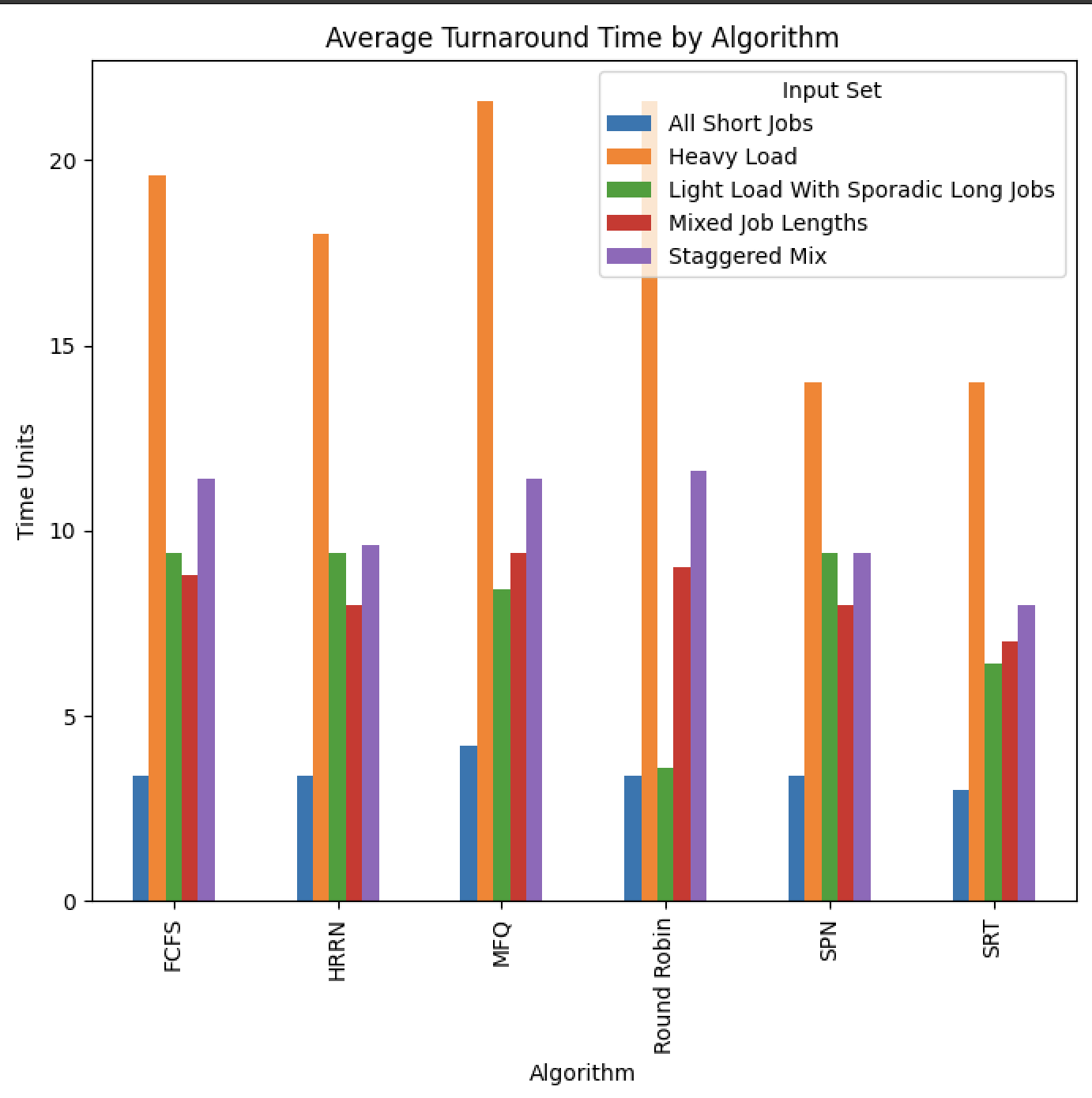
Table 1: Test Cases

| Input Set | Arrival Times | Service Times |
| --- | --- | --- |
| All Short Jobs | 0, 2, 4, 6, 8 | 2, 3, 2, 4, 1 |
| Mixed Job Lengths | 0, 1, 3, 5, 7 | 1, 8, 2, 7, 3 |
| Heavy Load | 0, 0, 0, 0, 0 | 10, 2, 8, 6, 4 |
| Light Load With Sporadic Jobs | 0, 5, 10, 15, 20 | 1, 12, 1, 12, 1 |
| Staggered Mix | 0, 3, 5, 8, 12 | 8, 2, 10, 1, 5 |

Table 2: Turnaround Time Results

| **Input Set** | **Algorithm** | **Average Turnaround Time** | **Average Waiting Time** |
| --- | --- | --- | --- |
| All Short Jobs | FCFS | 3.4 | 1.0 |
| All Short Jobs | Round Robin | 3.4 | 1.0 |
| All Short Jobs | SPN | 3.4 | 1.0 |
| All Short Jobs | SRT | 3.0 | 0.6 |
| All Short Jobs | HRRN | 3.4 | 1.0 |
| All Short Jobs | MFQ | 4.2 | 1.8 |
| Mixed Job Lengths | FCFS | 8.8 | 4.6 |
| Mixed Job Lengths | Round Robin | 9.0 | 4.8 |
| Mixed Job Lengths | SPN | 8.0 | 3.8 |
| Mixed Job Lengths | SRT | 7.0 | 2.8 |
| Mixed Job Lengths | HRRN | 8.0 | 3.8 |
| Mixed Job Lengths | MFQ | 9.4 | 5.2 |
| Heavy Load | FCFS | 19.6 | 13.6 |
| Heavy Load | Round Robin | 21.6 | 15.6 |
| Heavy Load | SPN | 14.0 | 8.0 |
| Heavy Load | SRT | 14.0 | 8.0 |
| Heavy Load | HRRN | 18.0 | 12.0 |
| Heavy Load | MFQ | 21.6 | 15.6 |
| Light Load With Sporadic Jobs | FCFS | 9.4 | 4.0 |
| Light Load With Sporadic Jobs | Round Robin | 3.6 | -1.8 |
| Light Load With Sporadic Jobs | SPN | 9.4 | 4.0 |
| Light Load With Sporadic Jobs | SRT | 6.4 | 1.0 |
| Light Load With Sporadic Jobs | HRRN | 9.4 | 4.0 |
| Light Load With Sporadic Jobs | MFQ | 8.4 | 3.0 |
| Staggered Mix | FCFS | 11.4 | 6.2 |
| Staggered Mix | Round Robin | 11.6 | 6.4 |
| Staggered Mix | SPN | 9.4 | 4.2 |
| Staggered Mix | SRT | 8.0 | 2.8 |
| Staggered Mix | HRRN | 9.6 | 4.4 |
| Staggered Mix | MFQ | 11.4 | 6.2 |

**Figure 3.18 Average Turnaround Time Comparison**

A graph of different colored bars

Description automatically generated

* For "All Short Jobs," algorithms like SRT showed improved performance with lower average turnaround times.
* "Mixed Job Lengths" saw better handling by SPN and SRT, demonstrating their efficiency with varied job lengths.
* Under "Heavy Load," SPN and SRT again performed well, indicating their effectiveness in high-load scenarios.
* "Light Load With Sporadic Long Jobs" highlighted discrepancies in the algorithms, with some like Round Robin showing negative waiting times, which indicates potential issues in time quantum settings.
* "Staggered Mix" presented a balanced workload where HRRN and SPN offered better average times.

The results underscore the importance of choosing the right algorithm for a given workload. SPN and SRT generally provided better average times across varied scenarios, indicating their robustness.

Code for analysis:

import pandas as pd

import matplotlib.pyplot as plt

from algorithms.fcfs import fcfs

from algorithms.hrrn import hrrn

from algorithms.srt import srt

from algorithms.spn import spn

from algorithms.rr import round\_robin

from algorithms.mfq import mlfq as mfq

from algorithms.custom import apsa

# Define the sample inputs

inputs = {

"All Short Jobs": {

"arrival\_times": [0, 2, 4, 6, 8],

"service\_times": [2, 3, 2, 4, 1],

},

"Mixed Job Lengths": {

"arrival\_times": [0, 1, 3, 5, 7],

"service\_times": [1, 8, 2, 7, 3],

},

"Heavy Load": {

"arrival\_times": [0, 0, 0, 0, 0],

"service\_times": [10, 2, 8, 6, 4],

},

"Light Load With Sporadic Long Jobs": {

"arrival\_times": [0, 5, 10, 15, 20],

"service\_times": [1, 12, 1, 12, 1],

},

"Staggered Mix": {

"arrival\_times": [0, 3, 5, 8, 12],

"service\_times": [8, 2, 10, 1, 5],

},

}

# Define the algorithms

algorithms = {

"FCFS": fcfs,

"Round Robin": round\_robin,

"SPN": spn,

"SRT": srt,

"HRRN": hrrn,

"MFQ": mfq,

}

# Define a time quantum for those algorithms that need it

time\_quantum1 = 4

time\_quantum2 = 8

# Prepare the results table

results\_data = {

"Input Set": [],

"Algorithm": [],

"Average Turnaround Time": [],

"Average Waiting Time": [],

}

# Run the algorithms for each input set and collect results

for input\_name, data in inputs.items():

for algo\_name, algo\_func in algorithms.items():

if algo\_name == "Round Robin":

waiting\_times, turnaround\_times = algo\_func(

data["arrival\_times"], data["service\_times"], time\_quantum1

)

elif algo\_name == "MFQ":

waiting\_times, turnaround\_times = algo\_func(

data["arrival\_times"],

data["service\_times"],

time\_quantum1,

time\_quantum2,

)

elif algo\_name == "APSA":

waiting\_times, turnaround\_times = algo\_func(

data["arrival\_times"], data["service\_times"], 0.5, 10

)

else:

waiting\_times, turnaround\_times = algo\_func(

data["arrival\_times"], data["service\_times"]

)

avg\_turnaround\_time = sum(turnaround\_times) / len(turnaround\_times)

avg\_waiting\_time = sum(waiting\_times) / len(waiting\_times)

results\_data["Input Set"].append(input\_name)

results\_data["Algorithm"].append(algo\_name)

results\_data["Average Turnaround Time"].append(avg\_turnaround\_time)

results\_data["Average Waiting Time"].append(avg\_waiting\_time)

# Convert to DataFrame for easy tabular display

results\_df = pd.DataFrame(results\_data)

print(results\_df)

# Plotting the results

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))

results\_df.pivot(

index="Algorithm", columns="Input Set", values="Average Turnaround Time"

).plot(kind="bar", ax=ax1)

ax1.set\_title("Average Turnaround Time by Algorithm")

ax1.set\_ylabel("Time Units")

results\_df.pivot(

index="Algorithm", columns="Input Set", values="Average Waiting Time"

).plot(kind="bar", ax=ax2)

ax2.set\_title("Average Waiting Time by Algorithm")

ax2.set\_ylabel("Time Units")

plt.tight\_layout()

plt.show()

**Conclusion:**

This project taps into several types of scheduling, such as FCFS, RR, SJF, SRT, HRRN, MFS and new algorithm. They are effectively coded in the Python language for the specified inputs. In conclusion, this project has been a journey into understanding how different scheduling methods manage computer tasks. By studying and creating these methods, we've learned about their strengths and weaknesses. The project highlights the importance of choosing the right method for the job, as it can make a big difference in efficiency. Our exploration opens opportunities for future improvements in how computers handle tasks, making them faster and more effective.

**References**

* 1. <https://www.geeksforgeeks.org/multilevel-feedback-queue-scheduling-mlfq-cpu-scheduling/>
  2. <https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm>