Table of Contents

1. Project Overview

The Scheduling Algorithms Simulator is a Python-based application designed to simulate and analyze various CPU scheduling algorithms commonly used in operating systems. The supported algorithms include First Come First Serve (FCFS), Shortest Job First (SJF), Shortest Remaining Time First (SRTF), Round Robin (RR), Earliest Deadline First (EDF), Priority Scheduling (both preemptive and non-preemptive), and Lottery Scheduling. The simulator aims to provide an educational tool for understanding how these algorithms manage processes by calculating key performance metrics and visualizing process execution through Gantt charts.

Key features of the project include:

- Simulation of 8 distinct scheduling algorithms.
- Calculation of performance metrics such as Turnaround Time (TAT), Waiting Time (WT), Response Time (RT), and Throughput.
- Visualization of process execution timelines via Gantt charts.
- User-friendly input system for process details and algorithm-specific parameters.

The project is designed with modularity in mind, separating algorithm implementations, utility functions, and the main program into distinct files for ease of maintenance and scalability.

2. Module-Wise Breakdown

The project is organized into the following modules:

- algorithms/: Directory containing individual implementations of each scheduling algorithm.
    - fcfs.py: Implements the First Come First Serve algorithm.
    - sjf.py: Implements the Shortest Job First algorithm (non-preemptive).
    - srtf.py: Implements the Shortest Remaining Time First algorithm (preemptive).
    - rr.py: Implements the Round Robin algorithm with a user-defined time quantum.
    - edf.py: Implements the Earliest Deadline First algorithm for real-time scheduling.
    - prio.py: Implements non-preemptive Priority Scheduling.

    - priop.py: Implements preemptive Priority Scheduling.
    - lottery.py: Implements Lottery Scheduling with ticket-based randomness.
- utils/: Directory containing utility modules for process management, metrics, and visualization.
    - process.py: Defines the Process class with attributes such as Process ID (PID), arrival time, burst time, priority, deadline, and tickets.
    - metrics.py: Provides functions to calculate and display performance metrics and process tables.
    - visualization.py: Handles Gantt chart generation using visualization libraries.
- main.py: The main entry point of the application, responsible for user interaction, algorithm selection, simulation execution, and result display.

3. Functionalities

The simulator offers the following core functionalities:

- Algorithm Selection: Users can select from 8 scheduling algorithms through a command-line interface.
- Process Input: Users provide the number of processes and their attributes (e.g., arrival time, burst time, priority, deadline, or tickets), tailored to the chosen algorithm.
- Metrics Calculation: The program computes:
    - Turnaround Time (TAT): Completion time minus arrival time.
    - Waiting Time (WT): Time spent waiting in the ready queue.
    - Response Time (RT): Time from arrival to first execution.
    - Throughput: Number of processes completed per unit time.
- Gantt Chart Visualization: Displays a timeline of process execution, including idle periods, using graphical charts.
- Modular Design: Each algorithm and utility function is encapsulated in its own module, enhancing code reusability and maintainability.
- Error Handling: Includes basic validation to handle invalid inputs and edge cases, such as empty process lists.

4. Technology Used

Programming Languages:

- Python: The primary language used for its simplicity, readability, and extensive library support.

Libraries and Tools:

- Matplotlib: Used to create Gantt charts for visualizing process execution timelines.
- Seaborn: Enhances the visual appeal of Gantt charts with modern styling options.
- NumPy: Supports numerical operations, particularly for managing chart colors and data arrays.
- Tabulate: Formats process tables for clear, readable console output.

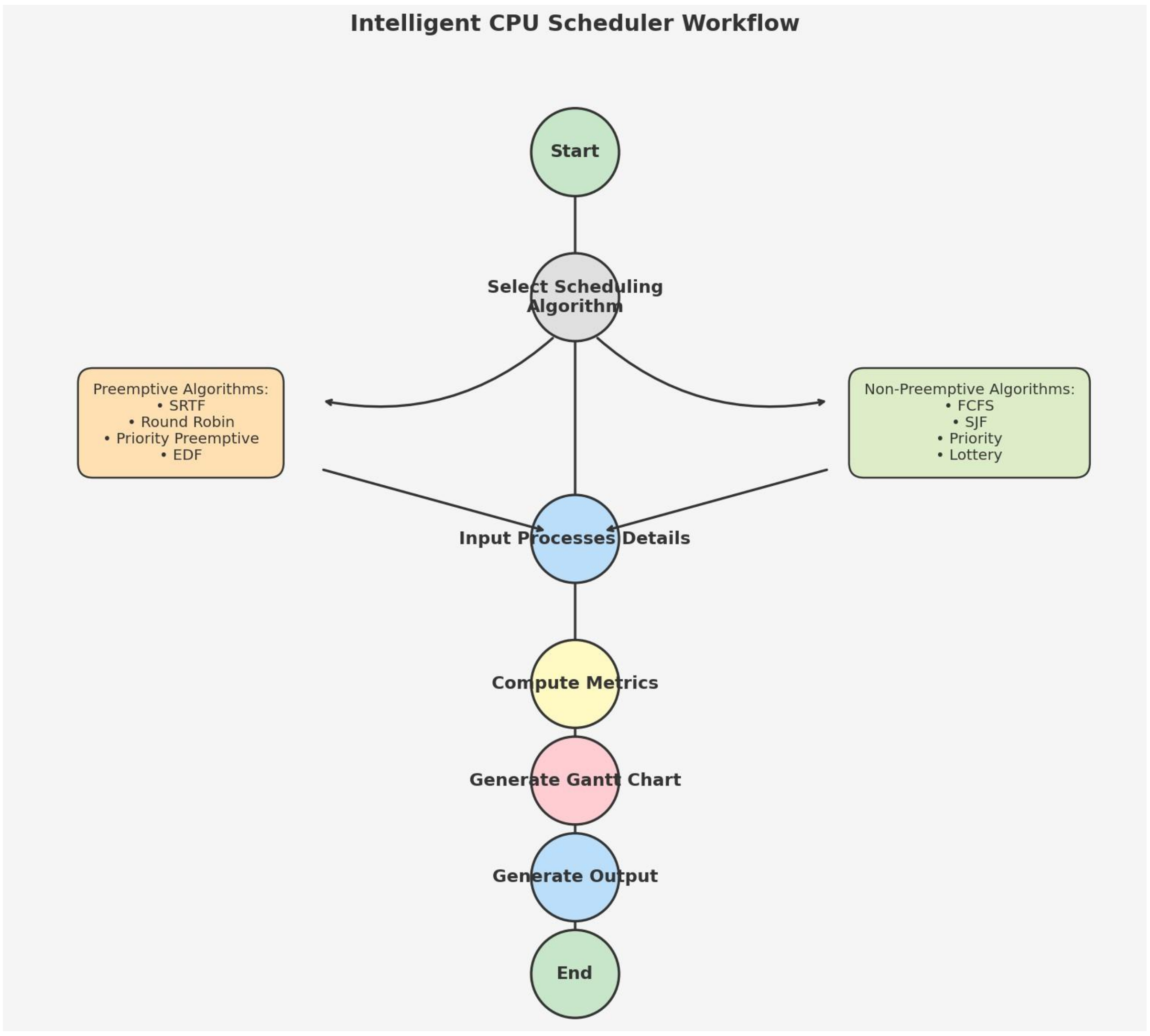
Other Tools:

- GitHub: Utilized for version control, collaboration, and hosting the project repository.

5. Flow Diagram

The following outlines the high-level flow of the simulator's execution:

1. Start: The user launches main.py.
2. Algorithm Selection: The user selects a scheduling algorithm from the provided options.
3. Input Processes: The user enters the number of processes and their details (e.g., arrival time, burst time).
4. Simulate Algorithm: The chosen algorithm processes the input and generates an execution timeline.
5. Calculate Metrics: Performance metrics (TAT, WT, RT, throughput) are computed based on the simulation.
6. Display Results: A process table and metrics are printed to the console.
7. Draw Gantt Chart: A visual representation of the execution timeline is generated and displayed.
8. End: The program terminates.



6. Revision Tracking on GitHub

- Repository Name: Process_Schedular_Simulator
- GitHub Link: Link

The project is tracked on GitHub, with key revisions including:

- Initial setup of project structure and files.
- Implementation of individual scheduling algorithms.
- Integration of metrics calculation and Gantt chart visualization.
- Enhancements to Lottery Scheduling for ticket-based execution.
- Bug fixes and performance optimizations based on testing.

7. Conclusion and Future Scope

The Scheduling Algorithms Simulator effectively demonstrates the behavior of various CPU scheduling algorithms, providing valuable insights into their performance through metrics and visualizations. It serves as an educational tool for students and a practical resource for understanding operating system concepts.

Future Scope:

- Additional Algorithms: Incorporate advanced algorithms like Multilevel Feedback Queue (MLFQ).
- Graphical User Interface (GUI): Replace the command-line interface with a more intuitive GUI.
- Real-Time Enhancements: Extend support for real-time scheduling scenarios.
- Comparative Analysis: Enable side-by-side comparison of multiple algorithms for the same process set.
- Testing Framework: Add unit tests to ensure algorithm correctness and robustness.

8. References

- Books:
  - Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts*. Wiley.
- Online Resources:
  - GeeksforGeeks: CPU Scheduling Algorithms
  - Matplotlib Documentation
  - Seaborn Documentation

Appendix

A. AI-Generated Project Elaboration/Breakdown Report

1. Architecture Overview

Modular Design:

The Scheduling Algorithms Simulator is designed using a modular architecture to promote maintainability and scalability. Implemented in Python, the project leverages Python's module system to achieve a clean and organized structure. The repository is structured as follows:

- algorithms/ Directory: Contains individual Python files for each scheduling algorithm, such as:
  - fcfs.py: Implements First-Come-First-Serve (FCFS) scheduling.
  - sjf.py: Implements Shortest Job First (SJF) scheduling.
  - rr.py: Implements Round Robin (RR) scheduling.
  - And similar files for SRTF, EDF, Priority Scheduling, and Lottery Scheduling.
    Each file encapsulates the logic for a specific algorithm, ensuring isolation and ease of extension.
- utils/ Directory: Houses utility modules that provide shared functionality:
  - process.py: Defines the Process class, which encapsulates process attributes (e.g., arrival time, burst time, priority) and computed metrics (e.g., completion time, waiting time).
  - metrics.py: Contains functions to calculate performance metrics such as turnaround time, waiting time, and response time.
  - visualization.py: Implements Gantt chart generation using libraries like Matplotlib and Seaborn for visualizing the scheduling timeline.
- main.py: The entry point of the simulator, responsible for user interaction, algorithm selection, and orchestrating the simulation workflow.
- README.md: Provides documentation on installation, usage instructions, and contribution guidelines.
- requirements.txt: Lists the Python dependencies (e.g., Matplotlib, Seaborn) required to run the project, ensuring easy setup across environments.

Design Principles:

- Separation of Concerns: Each module handles a distinct aspect of the simulator. Scheduling algorithms are isolated in their own files, while utility functions for process management, metrics calculation, and visualization are centralized in the utils/ directory, promoting clean and maintainable code.

- Extensibility: New scheduling algorithms can be added by creating a new file in the algorithms/ directory and integrating it into main.py without modifying existing code. This design allows for seamless expansion of the simulator's capabilities.
- Reusability: Utility functions, such as those in process.py, metrics.py, and visualization.py, are shared across all algorithms, reducing code duplication and enhancing efficiency.
- Cross-Platform Compatibility: As a Python-based tool, the simulator runs on any system with Python installed, making it inherently cross-platform. However, the visualization component requires a graphical environment, which may limit its use on headless systems unless adapted.

2. Workflow Overview

User Interaction Flow:

- Start the Simulator:
  The user launches the simulator by running main.py from the command line, initiating the interactive interface.
- Algorithm Selection:
  The user is presented with a list of supported scheduling algorithms (e.g., FCFS, SJF, SRTF, RR, EDF, Priority Scheduling, Lottery Scheduling) and prompted to select one.
- Input Process Details:
  The user specifies the number of processes and provides their details, which vary by algorithm:
  - For all algorithms: arrival time and burst time.
  - For Priority Scheduling: priority value.
  - For EDF (Earliest Deadline First): deadline.
  - For Lottery Scheduling: number of tickets.
  - For RR (Round Robin): time quantum.
    Input validation ensures that data is correctly formatted and appropriate for the chosen algorithm.
- Simulation:
  The selected algorithm processes the input data, simulating the execution of processes and generating a timeline that includes execution periods and any idle times.
- Metrics Calculation:
  Performance metrics (e.g., average turnaround time, average waiting time, response time) are calculated based on the simulation results, providing quantitative insights into the algorithm's efficiency.

- Display Results:
  The simulator outputs:
  - A formatted process table showing details like arrival time, burst time, completion time, and waiting time.
  - A summary of performance metrics.
  - A Gantt chart visualizing the execution timeline, generated via visualization.py.
- Feedback Loop:
  The simulator includes error handling for invalid inputs (e.g., non-existent algorithms, negative burst times) and provides clear console messages to guide the user. Results are displayed immediately, allowing users to assess the simulation and adjust inputs if needed.

B. Problem Statement

Develop a simulator to demonstrate the behavior of various CPU scheduling algorithms, including FCFS, SJF, SRTF, RR, EDF, Priority Scheduling (preemptive and non-preemptive), and Lottery Scheduling. The simulator should calculate key performance metrics (e.g., Turnaround Time, Waiting Time, Response Time, Throughput) and provide a visual representation of process execution via Gantt charts, enabling users to understand and compare algorithm performance.

C. Solution/Code

1. main.py

```
from utiLs.process import Process
from utiLs.metrics import print_aLgorithm_info,
caLcuLate_metrics
from utiLs.visuaLization import draw_gantt_chart
from aLgorithms.fcfs import fcfs
from aLgorithms.sjf import sjf
from aLgorithms.srtf import srtf
from aLgorithms.rr import round_robin
from aLgorithms.edf import earLiest_deadLine_first
from aLgorithms.prio import priority_scheduLing
from aLgorithms.priop import preemptive_priority_scheduLing
from aLgorithms.Lottery import Lottery_scheduLing

def get_processes(priority=FaLse, edf=FaLse, tickets=FaLse):
    processes = []
    n = int(input("Enter the number of processes: "))
    for i in range(1, n + 1):
        at = int(input(f"Enter ArrivaL Time for Process {i}
(defauLt 0): ") or 0)
        bt = int(input(f"Enter Burst Time for Process {i}: "))

        priority_vaL = None
        if priority:
            priority_vaL = int(input(f"Enter Priority for
Process {i} (Lower is higher priority, press Enter to skip for
Lottery): ") or None)

        deadLine = None
        if edf:
            deadLine = int(input(f"Enter DeadLine for Process
{i}: "))

        tickets_vaL = None
        if tickets:
            tickets_vaL = int(input(f"Enter number of tickets
for Process {i}: "))

        processes.append(Process(i, at, bt, priority_vaL,
deadLine, tickets_vaL))

    return processes

def main():
    aLgo = input("Choose ScheduLing ALgorithm (FCFS, SJF, SRTF,
RR, EDF, PRIO, PRIOP, LOTTERY): ").strip().upper()

    if aLgo  = "EDF":
        processes = get_processes(edf=True)
    eLif aLgo in ["PRIO", "PRIOP"]:
        processes = get_processes(priority=True)
    eLif aLgo  = "LOTTERY":
        processes = get_processes(tickets=True)
    eLse:
        processes = get_processes()

    print_aLgorithm_info(aLgo)

    if aLgo  = "FCFS":
        resuLt, timeLine = fcfs(processes)
    eLif aLgo  = "SJF":
        resuLt, timeLine = sjf(processes)
    eLif aLgo  = "SRTF":
        resuLt, timeLine = srtf(processes)
    eLif aLgo  = "RR":
        quantum = int(input("Enter time quantum: "))
        resuLt, timeLine = round_robin(processes, quantum)
    eLif aLgo  = "EDF":
        resuLt, timeLine = earLiest_deadLine_first(processes)
    eLif aLgo  = "PRIO":
        resuLt, timeLine = priority_scheduLing(processes,
preemptive=FaLse)
    eLif aLgo  = "PRIOP":
        resuLt, timeLine =
preemptive_priority_scheduLing(processes)
    eLif aLgo  = "LOTTERY":
        resuLt, timeLine = Lottery_scheduLing(processes)
    eLse:
        print("InvaLid choice!")
        return

    caLcuLate_metrics(resuLt, timeLine, aLgo)



    print("\nTimeLine before drawing Gantt chart:", timeLine)



    try:
        if timeLine:
            draw_gantt_chart(timeLine)
        eLse:
            print("Cannot draw Gantt chart: TimeLine is empty.")
    except Exception as e:
        print(f"Error drawing Gantt chart: {e}")

if  _name _  = " _main _":
    main()
```

2. utils/

a. Metrics

```
from tabuLate import tabuLate

def print_aLgorithm_info(aLgo):
    descriptions = {
        "FCFS": "First Come, First Serve (FCFS)
executes processes in order of arrivaL.",
        "SJF": "Shortest Job First (SJF) seLects the
process with the smaLLest burst time first.",
        "SRTF": "Shortest Remaining Time First (SRTF)
is a preemptive version of SJF.",
        "RR": "Round Robin (RR) gives each process a
fixed time sLice (quantum).",
        "EDF": "EarLiest DeadLine First (EDF)
prioritizes processes with the cLosest deadLines.",
        "PRIO": "Priority ScheduLing (Non-Preemptive)
runs the highest priority avaiLabLe process.",
        "PRIOP": "Priority ScheduLing (Preemptive)
aLLows priority-based preemption.",
        "LOTTERY": "Non-Preemptive Lottery ScheduLing
uses user-provided tickets for each process; a random
draw determines which process runs to compLetion."
    }
    print("\n" + "="*40)
    print(f"  {aLgo} ScheduLing ALgorithm")
    print("="*40)
    print(descriptions.get(aLgo, "InvaLid
ALgorithm"))
    print("="*40 + "\n")

def print_process_tabLe(processes, aLgo=""):

    sorted_processes = sorted(processes, key=Lambda
p: p.pid)


    if aLgo  = "LOTTERY":

        tabLe = [[
            p.pid, p.arrivaL_time, p.burst_time,
p.tickets,
            p.compLetion_time, p.turnaround_time,
p.waiting_time, p.response_time
        ] for p in sorted_processes]
        headers = ["PID", "AT", "BT", "Tickets",
"CT", "TAT", "WT", "RT"]
    eLse:

        tabLe = [[
            p.pid, p.arrivaL_time, p.burst_time,
p.priority if p.priority is not None eLse "-",
            p.compLetion_time, p.turnaround_time,
p.waiting_time, p.response_time
        ] for p in sorted_processes]
        headers = ["PID", "AT", "BT", "Priority",
"CT", "TAT", "WT", "RT"]

    print("\nProcess DetaiLs:\n")
    print(tabuLate(tabLe, headers=headers,
tabLefmt="fancy_grid"))

def caLcuLate_metrics(processes, timeLine, aLgo=""):
    print_process_tabLe(processes, aLgo)

    avg_tat = sum(p.turnaround_time for p in
processes) / Len(processes)
    avg_wt = sum(p.waiting_time for p in processes) /
Len(processes)
    avg_rt = sum(p.response_time for p in processes)
/ Len(processes)
    throughput = Len(processes) /
max(p.compLetion_time for p in processes)

    print("\n" + "="*30)
    print(f"Metrics Summary")
    print("="*30)
    print(f"Average Turnaround Time  :
{avg_tat:.2f}")
    print(f"Average  Waiting  Time   :  {avg_wt:.2f}")
    print(f"Average  Response  Time   :  {avg_rt:.2f}")
    print(f"Throughput                :
{throughput:.2f} processes/unit time")
    print("="*30 + "\n")
```

b. process.py

```
cLass Process:
    def  _init _(seLf, pid, arrivaL_time, burst_time,
priority=None, deadLine=None, tickets=None):
        seLf.pid  =  pid
        seLf.arrivaL_time = arrivaL_time
        seLf.burst_time = burst_time
        seLf.remaining_time = burst_time
        seLf.priority = priority
        seLf.deadLine = deadLine
        seLf.tickets = tickets
        seLf.start_time = -1
        seLf.compLetion_time = 0
        seLf.waiting_time = 0
        seLf.turnaround_time = 0
        seLf.response_time = -1
```

c. visualization.py

```
import matpLotLib.pypLot as pLt
import seaborn as sns
import numpy as np

def draw_gantt_chart(timeLine):

    sns.set_styLe("whitegrid")
    sns.set_context("notebook", font_scaLe=1.2)



    pLt.figure(figsize=(12, 3), dpi=100)



    paLette = sns.coLor_paLette("husL", 10)
    coLors = {}
    start = 0

    for pid, duration in timeLine:

        if pid not in coLors:
            if isinstance(pid, str) and pid.Lower()
 = "idLe":
                coLors[pid] =
sns.coLor_paLette("Greys", 1)[0]
            eLse:

                coLors[pid] = paLette[pid %
Len(paLette)]


        pLt.barh(y=0, width=duration, Left=start,
height=0.6, aLign='center',
                 coLor=coLors[pid],
edgecoLor="bLack", Linewidth=1.2)


        dispLay_name = f"P{pid}" if isinstance(pid,
int) eLse "IdLe"

        Luminance = 0.299 * coLors[pid][0] + 0.587 *
coLors[pid][1] + 0.114 * coLors[pid][2]
        text_coLor = 'white' if Luminance < 0.5 eLse
'bLack'

        pLt.text(start + duration / 2, 0,
dispLay_name, ha='center', va='center',
                 coLor=text_coLor, fontsize=10,
fontweight='boLd')

        start += duration


    pLt.xLabeL("Time", fontsize=12, LabeLpad=10)
    pLt.yLabeL("Processes", fontsize=12, LabeLpad=10)
    pLt.titLe("Gantt Chart", fontsize=14, pad=15)


    pLt.xticks(range(start + 1), fontsize=10)
    pLt.yticks([])


    pLt.grid(True, axis='x', LinestyLe=' -',
aLpha=0.7)


    pLt.tight_Layout()
    pLt.show()
```

d. __init__.py
   (Empty file to make directory a Python Package.)

3. algorithms/
   a. edf.py

```
import heapq
from utiLs.process import Process

def earLiest_deadLine_first(processes):
    processes.sort(key=Lambda p: p.arrivaL_time)
    ready_queue = []
    timeLine = []
    time, index, compLeted = 0, 0, 0

    whiLe compLeted < Len(processes):
        whiLe index < Len(processes) and 
processes[index].arrivaL_time  = time:
            heapq.heappush(ready_queue, 
(processes[index].deadLine, processes[index].pid, 
processes[index]))
            index += 1

        if ready_queue:
            _, _, p = heapq.heappop(ready_queue)
            if p.start_time  = -1:
                p.start_time = time
                p.response_time = time - 
p.arrivaL_time

            time += p.burst_time
            p.compLetion_time = time
            p.turnaround_time = p.compLetion_time - 
p.arrivaL_time
            p.waiting_time = p.turnaround_time - 
p.burst_time
            timeLine.append((p.pid, p.burst_time))
            compLeted += 1
        eLse:
            timeLine.append(("IdLe", 1))
            time += 1
    return processes, timeLine
```

b. fcfs.py

```
from utiLs.process import Process

def fcfs(processes):
    processes.sort(key=Lambda p: p.arrivaL_time)
    time = 0
    timeLine = []

    for p in processes:
        if time < p.arrivaL_time:
            timeLine.append(("IdLe", p.arrivaL_time -
time))
            time = p.arrivaL_time

        p.start_time = time
        p.response_time = time - p.arrivaL_time
        time += p.burst_time
        p.compLetion_time = time
        p.turnaround_time = p.compLetion_time -
p.arrivaL_time
        p.waiting_time = p.turnaround_time -
p.burst_time
        timeLine.append((p.pid, p.burst_time))

    return processes, timeLine
```

c. lottery.py

```
from utiLs.process import Process
import random
from coLLections import deque

def Lottery_scheduLing(processes):
    """
    Non-Preemptive Lottery ScheduLing ALgorithm (With
User-Provided Tickets):
    - Each process has a user-specified number of
tickets.
    - A random Lottery seLects the next process from
the ready queue.
```

```
    - The seLected process runs to compLetion without
interruption.
    - Processes with more tickets have a higher
chance of being seLected.
    """
    # Sort processes by arrivaL time to process
arrivaLs in order
    processes.sort(key=Lambda p: p.arrivaL_time)
    time = 0
    timeLine = []
    queue = deque()
    index = 0
    n = Len(processes)

    # Use the user-provided ticket counts
    ticket_counts = {}
    for p in processes:
        if p.tickets is None or p.tickets < 1:
            raise VaLueError(f"Process {p.pid} must
have a positive number of tickets for Lottery
ScheduLing.")
        ticket_counts[p.pid] = p.tickets

    # Main scheduLing Loop
    whiLe index < n or queue:
        # Add aLL processes that have arrived by the
current time to the queue
        whiLe index < n and
processes[index].arrivaL_time  = time:
            queue.append(processes[index])
            index += 1

        # If queue is empty, CPU is idLe untiL the
next process arrives
        if not queue:
            if index < n:
                idLe_time =
processes[index].arrivaL_time - time
                timeLine.append(("IdLe", idLe_time))
                time += idLe_time
                continue
            eLse:
                break  # No more processes to
scheduLe
```

```
        # Perform Lottery to seLect the next process
        totaL_tickets = sum(ticket_counts[p.pid] for
p in queue)
        winning_ticket = random.randint(0,
totaL_tickets - 1)
        cumuLative = 0
        winner = None
        for p in queue:
            tickets = ticket_counts[p.pid]
            cumuLative += tickets
            if winning_ticket < cumuLative:
                winner = p
                break

        # Remove the winner from the queue to execute
it
        queue.remove(winner)

        # Set start and response time if this is the
process's first execution
        if winner.start_time  = -1:
            winner.start_time = time
            winner.response_time = time -
winner.arrivaL_time

        # Run the process to compLetion
(non-preemptive)
        exec_time = winner.remaining_time
        time += exec_time
        winner.remaining_time -= exec_time
        timeLine.append((winner.pid, exec_time))

        # Update process metrics
        winner.compLetion_time = time
        winner.turnaround_time =
winner.compLetion_time - winner.arrivaL_time
        winner.waiting_time = winner.turnaround_time
- winner.burst_time

    return processes, timeLine
```

d. prio.py

```
import heapq
```

```
from utiLs.process import Process

def priority_scheduLing(processes, preemptive=FaLse):
    processes.sort(key=Lambda p: p.arrivaL_time)
    ready_queue = []
    timeLine = []
    time, index, compLeted = 0, 0, 0

    whiLe compLeted < Len(processes):
        whiLe index < Len(processes) and
processes[index].arrivaL_time  = time:
            heapq.heappush(ready_queue,
(processes[index].priority, processes[index].pid,
processes[index]))
            index += 1

        if ready_queue:
            _, _, p = heapq.heappop(ready_queue)

            if p.start_time  = -1:
                p.start_time = time
                p.response_time = time -
p.a rrivaL_time

            if preemptive:
                time += 1
                p.remaining_time -= 1
                timeLine.append((p.pid, 1))

                if p.remaining_time > 0:
                    heapq.heappush(ready_queue,
(p.priority, p.pid, p))
                eLse:
                    p.compLetion_time = time
                    p.turnaround_time =
p.compLetion_time - p.arrivaL_time
                    p.waiting_time =
p.turnaround_time - p.burst_time
                    compLeted += 1
            eLse:
                time += p.burst_time
                p.compLetion_time = time
                p.turnaround_time = p.compLetion_time
- p.arrivaL_time
                p.waiting_time = p.turnaround_time -
```

```
p.burst_time
                timeLine.append((p.pid, 
p.burst_time))
                compLeted += 1
        eLse:
            timeLine.append(("IdLe", 1))
            time += 1

    return processes, timeLine
```

e. priop.py

```
from .prio import priority_scheduLing

def preemptive_priority_scheduLing(processes):
    return priority_scheduLing(processes, 
preemptive=True)
```

f. rr.py

```
from coLLections import deque
from utiLs.process import Process  # Assuming a 
Process cLass exists

def round_robin(processes, quantum=2):
    # Ensure quantum is vaLid
    if quantum  = 0:
        print("quantum must be greater than 0. 
Setting defauLt to 2.")
        quantum = 2

    # Sort processes by arrivaL time
    processes.sort(key=Lambda p: p.arrivaL_time)

    time = 0  # Current time
    timeLine = []  # Gantt chart timeLine
    queue = deque()  # queue for ready processes
    index = 0  # Index to track unprocessed processes
    n = Len(processes)

    whiLe index < n or queue:
        # Add aLL processes that have arrived by the 
current time
```

```
        whiLe index < n and
processes[index].arrivaL_time  = time:
            queue.append(processes[index])
            index += 1

        # If queue is empty but processes remain, CPU
is idLe
        if not queue and index < n:
            next_arrivaL =
processes[index].arrivaL_time
            idLe_time = next_arrivaL - time
            timeLine.append(("IdLe", idLe_time))
            time = next_arrivaL
            continue

        # If no more processes and queue is empty,
exit
        if not queue:
            break

        # Process the next process in the queue
        p = queue.popLeft()

        # Set start and response time if this is the
first execution
        if p.start_time  = -1:
            p.start_time = time
            p.response_time = time - p.arrivaL_time

        # Execute for quantum or remaining time,
whichever is smaLLer
        exec_time = min(p.remaining_time, quantum)
        time += exec_time
        p.remaining_time -= exec_time
        timeLine.append((p.pid, exec_time))

        # Add any new processes that arrived during
execution
        whiLe index < n and
processes[index].arrivaL_time  = time:
            queue.append(processes[index])
            index += 1

        # If process isn't finished, put it back in
the queue
```

```
        if p.remaining_time > 0:
            queue.append(p)
        eLse:
            # Process is compLete, caLcuLate metrics
            p.compLetion_time = time
            p.turnaround_time = p.compLetion_time -
p.arrivaL_time
            p.waiting_time = p.turnaround_time -
p.burst_time

    return processes, timeLine
```

g. sjf.py

```
import heapq
from utiLs.process import Process

def sjf(processes):
    processes.sort(key=Lambda p: (p.arrivaL_time,
p.burst_time))
    ready_queue = []
    time, index, compLeted = 0, 0, 0
    timeLine = []

    whiLe compLeted < Len(processes):
        whiLe index < Len(processes) and
processes[index].arrivaL_time  = time:
            heapq.heappush(ready_queue,
(processes[index].burst_time, processes[index].pid,
processes[index]))
            index += 1

        if ready_queue:
            _, _, p = heapq.heappop(ready_queue)
            if p.start_time  = -1:
                p.start_time = time
                p.response_time = time -
p.a rrivaL_time

            time += p.burst_time
            p.compLetion_time = time
            p.turnaround_time = p.compLetion_time -
p.arrivaL_time
            p.waiting_time = p.turnaround_time -
```

```
p.b urst_time
            timeLine.append((p.pid, p.burst_time))
            compLeted += 1
        eLse:
            timeLine.append(("IdLe", 1))
            time += 1

    return processes, timeLine
```

h. srtf.py

```
import heapq
from utiLs.process import Process

def srtf(processes):
    processes.sort(key=Lambda p: p.arrivaL_time)
    ready_queue = []
    timeLine = []
    time, index, compLeted = 0, 0, 0

    whiLe compLeted < Len(processes):
        whiLe index < Len(processes) and
processes[index].arrivaL_time  = time:
            heapq.heappush(ready_queue,
(processes[index].remaining_time,
processes[index].pid, processes[index]))
            index += 1

        if ready_queue:
            _, _, p = heapq.heappop(ready_queue)
            if p.start_time  = -1:
                p.start_time = time
                p.response_time = time -
p.a rrivaL_time

            p.remaining_time -= 1
            timeLine.append((p.pid, 1))
            time += 1

            if p.remaining_time > 0:
                heapq.heappush(ready_queue,
(p.remaining_time, p.pid, p))
            eLse:
                p.compLetion_time = time
                p.turnaround_time = p.compLetion_time
- p.arrivaL_time
                p.waiting_time = p.turnaround_time -
p.b urst_time
                compLeted += 1
        eLse:
            timeLine.append(("IdLe", 1))
            time += 1

    return processes, timeLine
```

i. __init__.py
   (Empty file to make directory a Python Package.)