ACM369 TERM PROJECT

Title: Processes Tasks and Implementations in Topics Covered in Our Course Involving the Python Virtual Machine — Enhanced with Artificial Intelligence

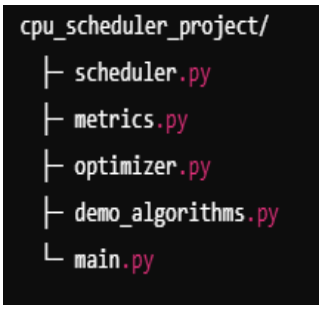
Name: Taylan Özveren

Introduction & Project Overview

This project brings together two main themes from Dec 369 proposed topics: Process tasks and applications (Topic #8) and Artificial Intelligence (Topic #10). This study shows how modern computing systems can integrate basic Operating System concepts with artificial intelligence-assisted optimization. Traditionally, operating systems manage processes and tasks with well-known scheduling algorithms such as FCFS, Round Robin, Priority Scheduling. However, recent developments in Artificial Intelligence have the potential to improve system efficiency and performance by providing opportunities to improve or even change some strategies at the OS level.

A.) Application Part and if necessary Programming Part

Project Structure and Summary Descriptions;



1.) **scheduler.py (CPU scheduling algorithms: OS core)**
2.) **metrics.py (performance metrics and how they are computed)**
3.) **demo_algorithms.py (mini demonstration of each algorithm, possibly with Gantt charts)**
4.) **optimizer.py (Genetic Algorithm for AI-based scheduling optimization)**
5.) **main.py (comprehensive test and final comparison, bridging OS and AI)**

1.) Scheduler.py

This file forms the center of the classic CPU scheduling within the scope of the project. It defines five basic algorithms: Non-preemptive (FCFS, SJF, Priority) and preemptive (Round Robin, Preemptive SJF).

- **OS Perspective: Showcases how an operating system (or our Python-based simulation) handles CPU time allocation to multiple processes.**
- **Project Integration: Forms the baseline for comparing and contrasting with AI-driven optimization (Genetic Algorithms).**
- **Educational Value: Each algorithm highlights key scheduling concepts taught in an OS course**

taylanozveren@vbox:~/cpu_scheduler_project — nano scheduler.py

GNU nano 7.2 scheduler.py

```
# scheduler.py
def fcfs(processes):
    """
    Non-preemptive FCFS:
    Returns a list of waiting times for each process in 'processes'.
    """
    # Prosesleri arrival_time'a göre sırala
    procs = sorted(processes, key=lambda x: x['arrival_time'])
    current_time = 0
    waiting_times = []

    for p in procs:
        if current_time < p['arrival_time']:
            current_time = p['arrival_time']
        waiting_time = current_time - p['arrival_time']
        waiting_times.append(waiting_time)
        current_time += p['burst_time']

    return waiting_times

def sjf(processes):
    """
    Non-preemptive SJF:
    Returns a list of waiting times.
    """
    # arrival_time ve burst_time'a göre sıralayalım
    procs = sorted(processes, key=lambda x: (x['arrival_time'], x['burst_time']))
    current_time = 0
    waiting_times = []

    for p in procs:
        if current_time < p['arrival_time']:
            current_time = p['arrival_time']
        waiting_time = current_time - p['arrival_time']
        waiting_times.append(waiting_time)
        current_time += p['burst_time']

    return waiting_times

def round_robin(processes, quantum=4):
    """
    Preemptive Round Robin:
    Returns a list of waiting times for each 'id'
```

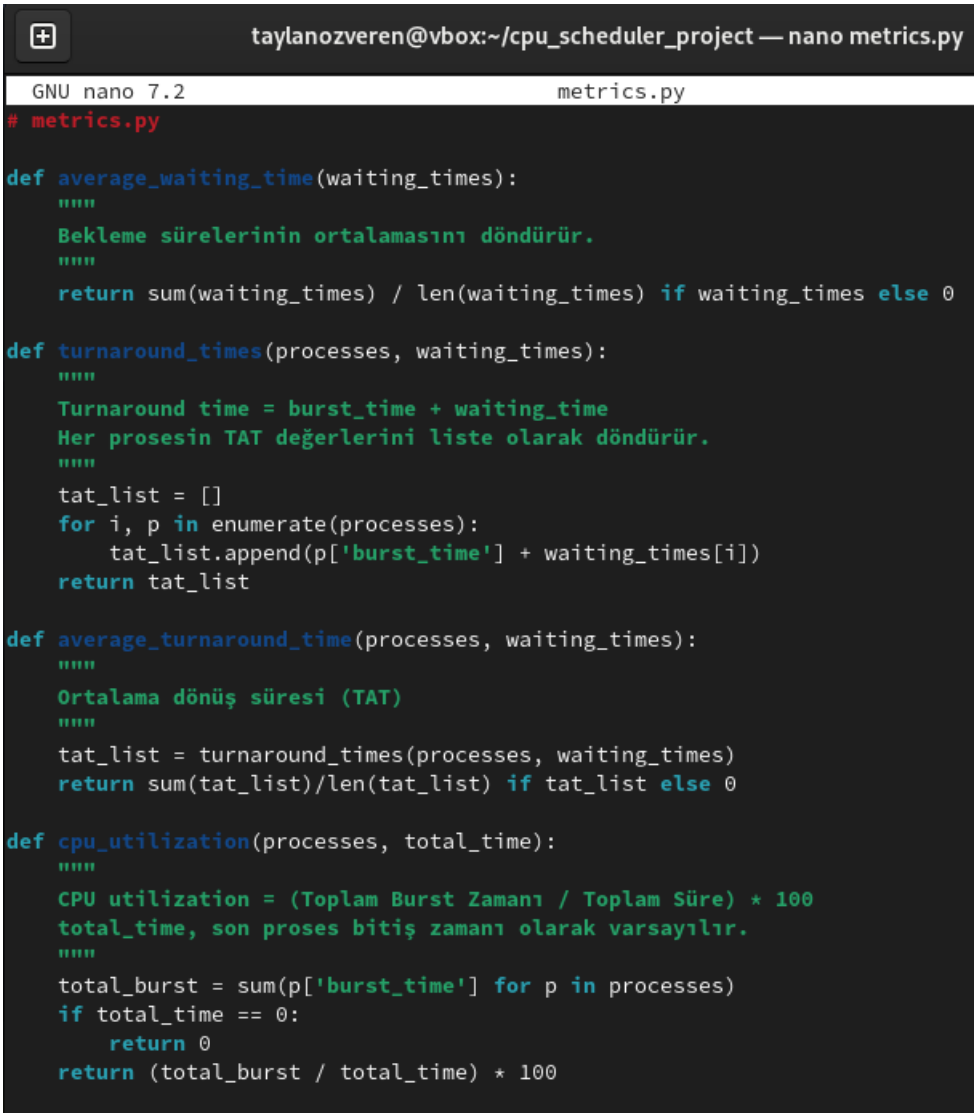
taylanozveren@vbox:~/cpu_scheduler_project — nano scheduler.py

GNU nano 7.2 scheduler.py

```
    from copy import deepcopy
    procs = deepcopy(processes)
    # arrival_time'a göre sırala (kolaylık için)
    procs = sorted(procs, key=lambda x: x['arrival_time'])

    n = len(procs)
    waiting_times = [0]*n
    remaining_burst = [p['burst_time'] for p in procs]

    completed = 0
    current_time = 0
    min_burst_index = -1
    check = False
    inf = 999999999

    while completed != n:
        min_burst = inf
        for i in range(n):
            if (procs[i]['arrival_time'] <= current_time) and (remaining_burst[i] < min_bur>
                min_burst = remaining_burst[i]
                min_burst_index = i
                check = True

        if not check:
            current_time += 1
            continue

        remaining_burst[min_burst_index] -= 1
        min_burst = remaining_burst[min_burst_index] if remaining_burst[min_burst_index] > >

        if remaining_burst[min_burst_index] == 0:
            completed += 1
            finish_time = current_time + 1
            # waiting_time = finish_time - arrival_time - burst_time
            waiting_times[procs[min_burst_index]['id']] = finish_time - procs[min_burst_ind>
            if waiting_times[procs[min_burst_index]['id']] < 0:
                waiting_times[procs[min_burst_index]['id']] = 0

        current_time += 1
        check = False

    return waiting_times
```

2.) Metrics.py

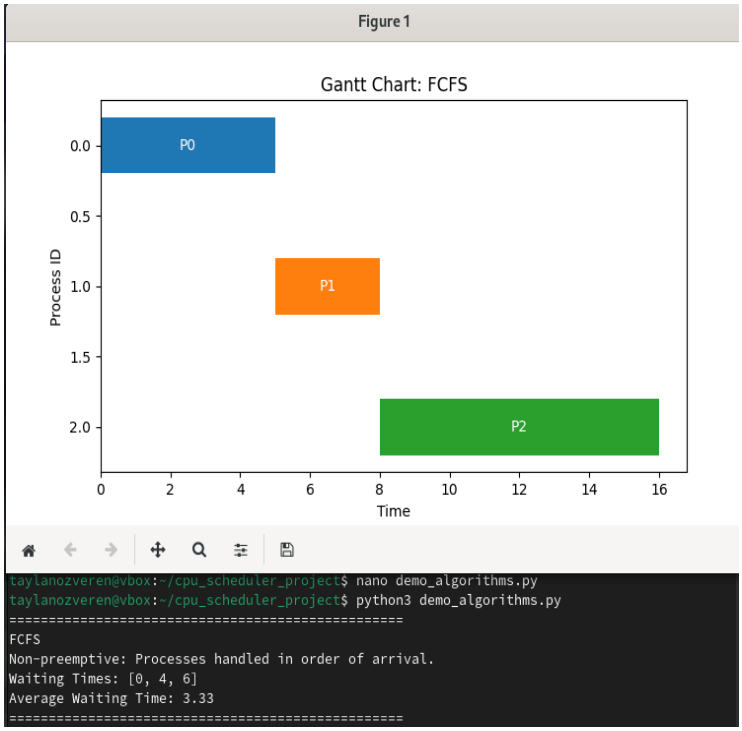
The metrics.py file it provides basic functions for evaluating the performance of CPU timing algorithms in operating systems. It calculates critical metrics such as average standby time, turnaround time, and CPU utilization rate. These metrics are vital for analyzing the efficiency of scheduling policies. It helps to measure how well system resources are allocated and how efficiently processes are managed, which makes it an important component for performance evaluation in scheduling systems.

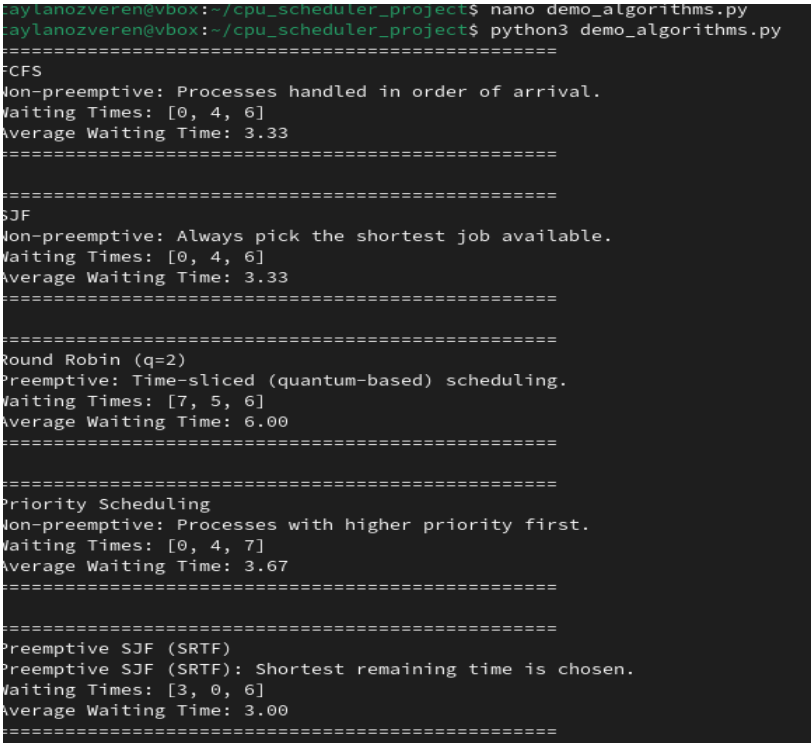


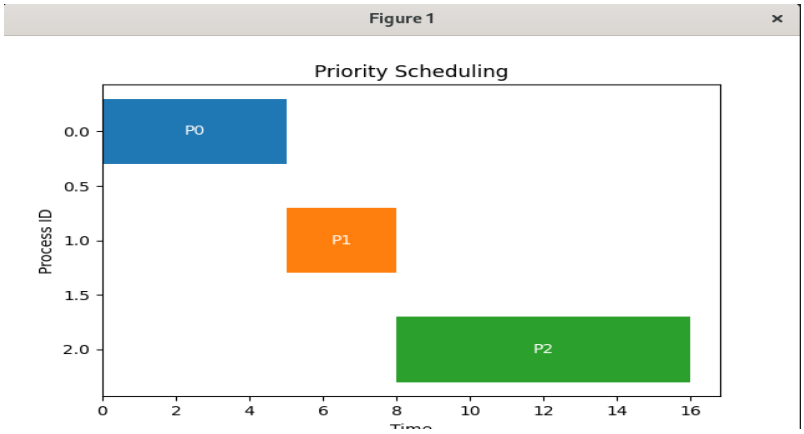
3.) Demo_scheduling.py

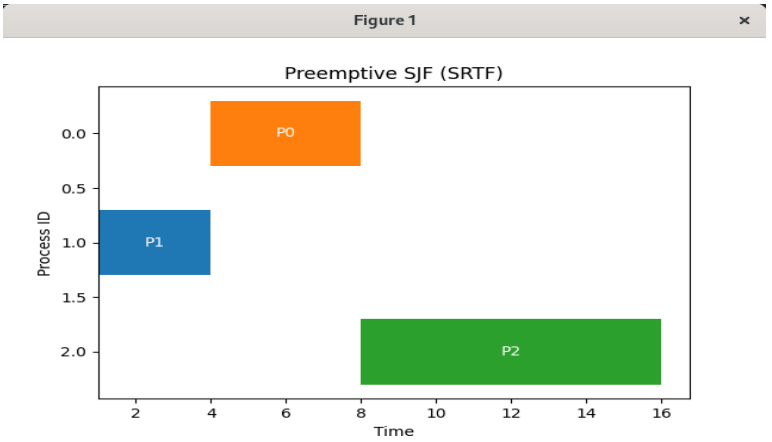
Emo_algorithms.py the script runs each scheduling algorithm on a simple list of transactions and shows the waiting times and the average waiting time for each transaction. It also includes a basic Gantt graph generation function for FCFS (simulate_fcfs_for_gantt and draw_gantt_chart_fcfs). This graph visually shows when each process starts and ends using colored blocks—the blue, orange, and green blocks represent the CPU usage of the processes.

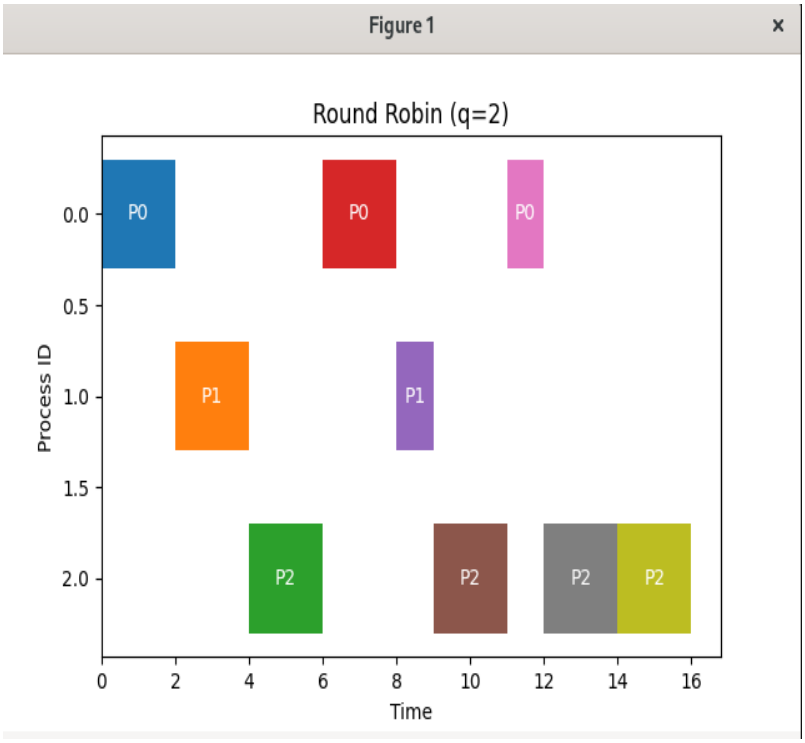
3.1) Demo_Algorithms.py OUTPUT IMAGE;

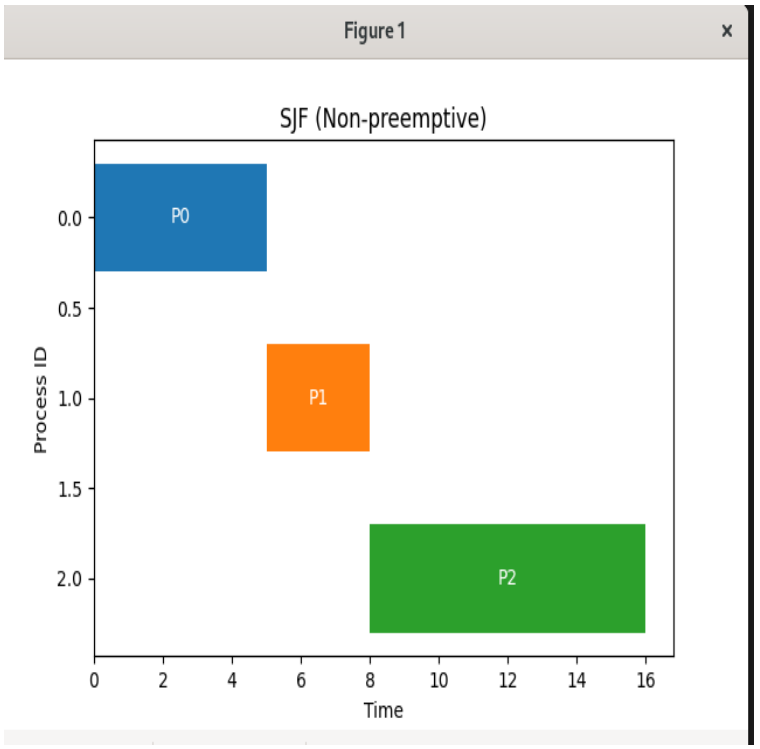












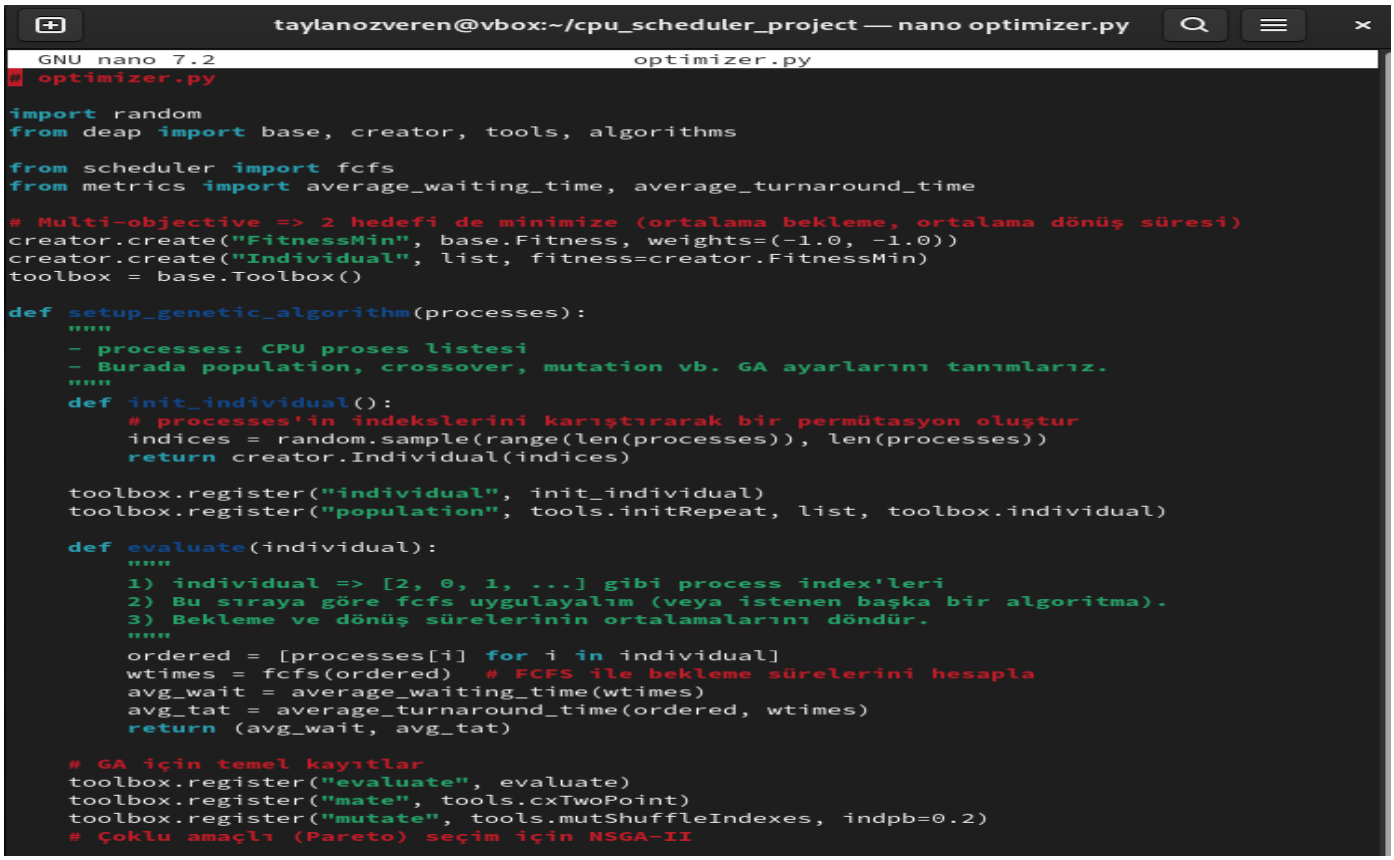
The output images for each scheduling algorithm clearly show the Waiting Times and Average Waiting Time (AWT) values in a regular and readable format. Also, Gantt Charts for all algorithms represent transaction execution timelines by visually presenting when processes start and finish on the processor.

These results were generated with detailed Gantt Charts created using matplotlib. Each graph highlights the behavior of the corresponding algorithm:

- **FCFS executes the transactions sequentially according to the arrival time.**
- **SJF sorts transactions by processing time to minimize the waiting time.**
- **The Round Robin divides the execution into time zones (quantum).**
- **Priority Scheduling reflects the order of execution according to the assigned priorities.**
- **Preemptive SJF (SRTF) dynamically interrupts and reschedules according to the remaining processing time.**

4.) Optimizer.py

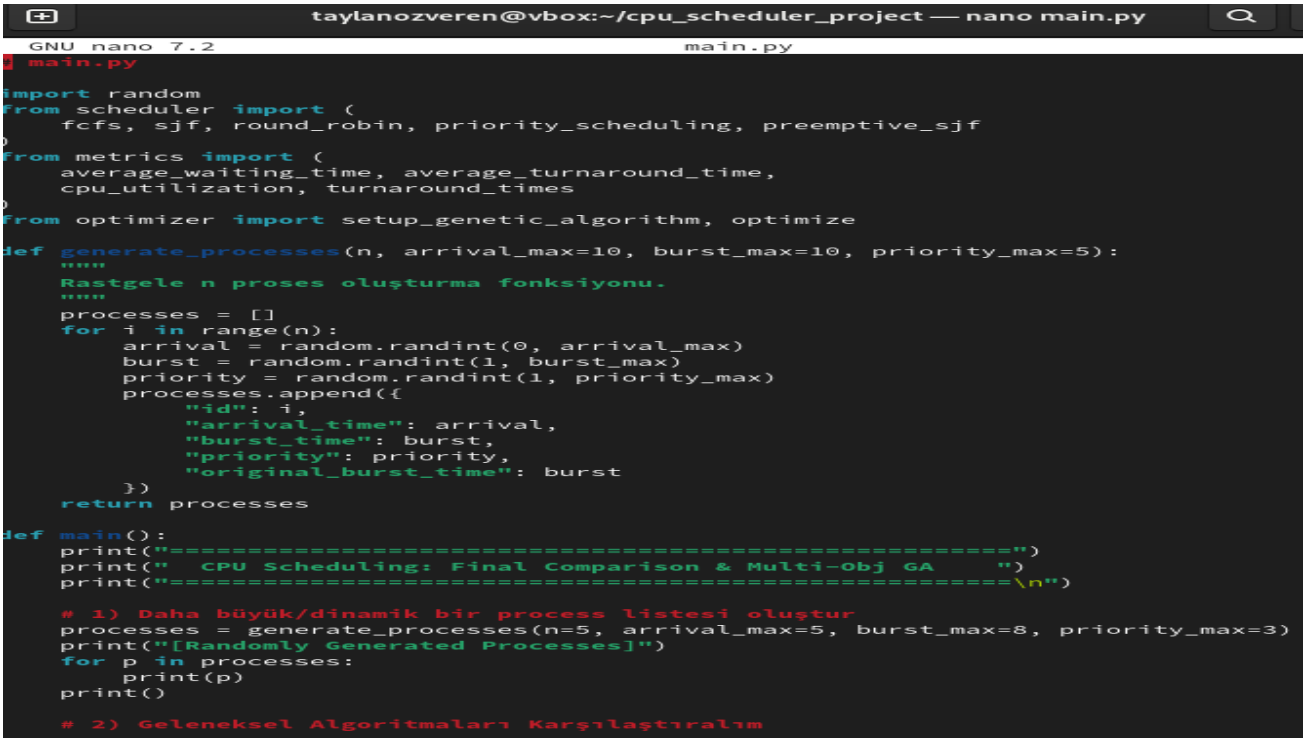
The optimizer.py file integrates both operating systems and optimization concepts by using Genetic Algorithms (GAs) to enhance CPU scheduling. While technically still scheduling, it goes beyond traditional OS methods by applying crossover, mutation, and selection to find optimal or near-optimal process orderings.



```python
    # Çoklu amaçlı (Pareto) seçim için NSGA-II
    toolbox.register("select", tools.selNSGA2)

    return toolbox

def optimize(toolbox, ngen=40, mu=50, lambda_=50):
    """
    Multi-objective GA (NSGA-II) çalıştırır:
    - mu: popülasyon boyutu
    - lambda_: her jenerasyonda üretilecek yeni birey sayısı
    - ngen: jenerasyon sayısı
    """
    # Başlangıç popülasyonunu oluştur
    pop = toolbox.population(n=mu)

    # İlk olarak pop'taki bireylerin fitness değerlerini hesaplayalım
    for ind in pop:
        ind.fitness.values = toolbox.evaluate(ind)

    # NSGA-II sistemi için başlangıçta selNSGA2 ile pop sıralanır
    pop = tools.selNSGA2(pop, len(pop))

    # Daha üst seviye bir evrim fonksiyonu (mu+lambda) tipi
    # eaMuPlusLambda(pop, toolbox, mu, lambda_, cxpb, mutpb, ngen...)
    pop, _ = algorithms.eaMuPlusLambda(
        population=pop,
        toolbox=toolbox,
        mu=mu,
        lambda_=lambda_,
        cxpb=0.7,       # crossover olasılığı
        mutpb=0.2,      # mutasyon olasılığı
        ngen=ngen,
        stats=None,
        halloffame=None,
        verbose=False
    )

    # En iyi çözümler (Pareto front) pop içerisinde bulunur
    return pop
```

5.) Main.py

Main.py the file functions as the "control center" of the project and combines both the classic operating system scheduling algorithms (F Decfs, SJF, Round Robin, Priority, Preemptive SJF) and the Genetic Algorithm for artificial intelligence-assisted optimization. At each run, a new randomly generated set of operations is generated and evaluated first by traditional algorithms, after which GA tries to find a more optimal scheduling sequence. In this process, the performance of traditional methods and how effective the artificial intelligence-supported results are are comprehensively compared. This shows in the clearest way that operating system principles meet with machine intelligence to find efficient scheduling solutions.



taylanozveren@vbox:~/cpu_scheduler_project — nano main.py

GNU nano 7.2 main.py

```
    print("[Traditional CPU Scheduling Algorithms]\n")

    # FCFS
    fcfs_wait = fcfs(processes)
    fcfs_avg_wait = average_waiting_time(fcfs_wait)
    fcfs_avg_tat = average_turnaround_time(processes, fcfs_wait)
    total_time = max(p['arrival_time'] + p['burst_time'] for p in processes)
    fcfs_cpu = cpu_utilization(processes, total_time)
    print(f"FCFS => Avg Wait: {fcfs_avg_wait:.2f}, Avg TAT: {fcfs_avg_tat:.2f}, CPU Util: {

    # SJF
    sjf_wait = sjf(processes)
    sjf_avg_wait = average_waiting_time(sjf_wait)
    sjf_avg_tat = average_turnaround_time(processes, sjf_wait)
    print(f"SJF  => Avg Wait: {sjf_avg_wait:.2f}, Avg TAT: {sjf_avg_tat:.2f}")

    # RR
    rr_wait = round_robin(processes, quantum=3)
    rr_avg_wait = average_waiting_time(rr_wait)
    rr_avg_tat = average_turnaround_time(processes, rr_wait)
    print(f"RR(q=3) => Avg Wait: {rr_avg_wait:.2f}, Avg TAT: {rr_avg_tat:.2f}")

    # Priority
    prio_wait = priority_scheduling(processes)
    prio_avg_wait = average_waiting_time(prio_wait)
    prio_avg_tat = average_turnaround_time(processes, prio_wait)
    print(f"Priority => Avg Wait: {prio_avg_wait:.2f}, Avg TAT: {prio_avg_tat:.2f}")

    # Preemptive SJF
    psjf_wait = preemptive_sjf(processes)
    psjf_avg_wait = average_waiting_time(psjf_wait)
    psjf_avg_tat = average_turnaround_time(processes, psjf_wait)
    print(f"Preemptive SJF => Avg Wait: {psjf_avg_wait:.2f}, Avg TAT: {psjf_avg_tat:.2f}")

    print("\n[Genetic Algorithm: Multi-Objective (Avg Wait, Avg TAT)]\n")

    # 3) Genetik Algoritma Ayarları
    toolbox = setup_genetic_algorithm(processes)
    # optimize fonksiyonu multi-objective NSGA2 ile çalışır
    pareto_pop = optimize(toolbox, ngen=20, mu=10, lambda_=10)

    # 4) Pareto-front'tan ilk birkaç bireyi inceleyelim
    # Tools içindeki 'sortNondominated' ile nondominated sorting yapabiliriz.
```

```
    from deap.tools import sortNondominated
    fronts = sortNondominated(pareto_pop, k=len(pareto_pop), first_front_only=False)
    best_front = fronts[0]  # Pareto'nun en üst front'u

    print(f"Found {len(best_front)} solutions in the first Pareto front.")
    print("Showing up to 3 solutions (AvgWait, AvgTAT) and their ordering:\n")

    for i, ind in enumerate(best_front[:3]):
        print(f"Solution #{i+1} -> Fitness(AvgWait, AvgTAT) = {ind.fitness.values}")
        print(f"Order: {ind}\n")

    print("===== End of main.py =====")

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

MAIN.PY OUTPUT IMAGE;

```
taylanozveren@vbox:~/cpu_scheduler_project$ python3 main.py
============================================================
  CPU Scheduling: Final Comparison & Multi-Obj GA
============================================================

[Randomly Generated Processes]
{'id': 0, 'arrival_time': 1, 'burst_time': 8, 'priority': 2, 'original_burst_time': 8}
{'id': 1, 'arrival_time': 1, 'burst_time': 3, 'priority': 2, 'original_burst_time': 3}
{'id': 2, 'arrival_time': 5, 'burst_time': 2, 'priority': 1, 'original_burst_time': 2}
{'id': 3, 'arrival_time': 1, 'burst_time': 7, 'priority': 3, 'original_burst_time': 7}
{'id': 4, 'arrival_time': 4, 'burst_time': 4, 'priority': 3, 'original_burst_time': 4}

[Traditional CPU Scheduling Algorithms]

FCFS => Avg Wait: 10.40, Avg TAT: 15.20, CPU Util: 266.67%
SJF  => Avg Wait: 9.20, Avg TAT: 14.00
RR(q=3) => Avg Wait: 9.20, Avg TAT: 14.00
Priority => Avg Wait: 11.60, Avg TAT: 16.40
Preemptive SJF => Avg Wait: 5.40, Avg TAT: 10.20

[Genetic Algorithm: Multi-Objective (Avg Wait, Avg TAT)]

Found 10 solutions in the first Pareto front.
Showing up to 3 solutions (AvgWait, AvgTAT) and their ordering:

Solution #1 -> Fitness(AvgWait, AvgTAT) = (2.4, 4.6)
Order: [2, 2, 2, 1, 2]

Solution #2 -> Fitness(AvgWait, AvgTAT) = (2.4, 4.6)
Order: [2, 2, 2, 1, 2]

Solution #3 -> Fitness(AvgWait, AvgTAT) = (2.4, 4.6)
Order: [2, 2, 2, 1, 2]

===== End of main.py =====
```

Main.py Description of the Output;

[Randomly Generated Processes]: The details of the five randomly generated processes are listed as {id, arrival_time, burst_time, priority}.

[Traditional CPU Timing Algorithms]: The following metrics are shown for each traditional scheduling algorithm (FCFS, SJF, Round Robin, Priority, Preemptive SJF)::
Average Waiting Time (Avg Wait) Average Return Time (Avg TAT) Simple CPU Utilization Rate
These results describe how classical algorithms perform on the existing randomly generated list of operations.

[Genetic Algorithm: Multipurpose (Avg Wait, Avg TASTE)] The Genetic Algorithm (GA) is executed and the first Pareto front is found.

The phrase "Found X solutions in the first Pareto front" states that multiple optimal solutions are equally good, and each provides different balances. For example, "Solution #1 -> Fitness(AvgWait, AvgTAT) = (2.4, 4.6)" indicates that the average standby time of this solution is 2.4 and the average return time is 4.6. The corresponding order of operations is, [2, 2, 2, 1, 2] it can be like and represents the planning sequence of operations.

B.) Report Part

Description of the Study

This project deals with CPU timing algorithms in operating systems and aims to improve these algorithms with Genetic Algorithms (GA). Classical methods such as FCFS, SJF, Round Robin, Priority Scheduling and Preemptive SJF were used, these methods were evaluated with metrics such as average standby time (AWT), average return time (TAT) and CPU utilization rate. Then, NSGA-II based Genetic Algorithms were integrated into the project in order to minimize AWT and TAT simultaneously. Thus, it is aimed to obtain more effective scheduling solutions by presenting traditional and artificial intelligence-based techniques Decoupled together.

Project Components

- **Scheduler.py**

It is the file where the classical CPU timing algorithms (FCFS, SJF, Round Robin, Priority, Preemptive SJF) are encoded. Each algorithm allows us to perform basic performance analysis by calculating the waiting times of processes.

- **Metrics.py**

It is the file where metric calculations are made. Here, values such as average standby time, average return time and CPU utilization rate are calculated under one roof. This ensures consistency and the outputs of the algorithms can be compared to a common measurement standard.

- **Demo_Algorithms.py**

It was created to clarify how each algorithm works on a small sample. The code visualizes the start and end times of processes using Gantt charts. In this way, the timeline of non-preemptive algorithms, especially FCFS, can be observed.

- **Optimizer.py**

It is a Genetic Algorithm module. Using evolutionary processes such as crossover, mutation, and selection, it tries to find the best or multi-goal balanced (Pareto front) order of operations. The ”fitness" values here are aimed at minimizing AWT and TAT at the same time in the project.

- **Main.py**

This is the “orchestrator” file of the project. Generates random process lists, applies these lists to classical scheduling algorithms, reports the results. Then it runs the Genetic Algorithm and provides the opportunity to compare the output of multi-objective (AWT + TAT) optimization.

Challenges and Solutions Encountered

Multi-Purpose Optimization

The challenge: Minimizing both AWT and TAT together is more complicated than a single metric.

Solution: Using the NSGA-II method, we sorted the different solutions on the “Pareto front”; thus, we determined which solution might be more suitable in different scenarios.

Visualizing Scheduling Processes

Difficulty: Interrupts and restarts, especially in preemptive algorithms, can extract a timeline whose visual expression is difficult.

Solution: Additional simulation functions were written to create Gantt Oct charts with Matplotlib and show the start/end times of processes piece by piece.

Extra challenge&solution; I had a constant optimization problem while drawing gant charts in the visualization part, and it made me difficult to show them all at the same time, thanks to the python libraries and ai supports, I also solved this problem

Inputs and Outputs

Entered:

user-defined or randomly generated process lists with {id, arrival_time, burst_time, priority} properties.

They Came Out:

The average waiting time of each algorithm, the average return time and the (simplified) CPU utilization rate.

Gantt graphs visualize the scheduling scheme of different algorithms, especially in the demo file.

GA results: Pareto front solutions obtained with NSGA-II show the values of each solution (AWT, TAT) and the process order on the screen.

The Function and Importance of the Project

This project presents a practical application of the CPU timing algorithms taught in the operating systems course. At the same time, it also shows how single or multiple metric optimization can be performed by using artificial intelligence techniques such as Genetic Algorithm. The study of traditional and innovative approaches together, as well as theoretical learning (e.g. Dec. The convoy effect of FCFS provides valuable experience in terms of finding solutions to real-world problems (AI optimization) as well as the time-sharing structure of Round Robin, the starvation problem in Priority).

In this way, not only can the operating system design be dependent on classical approaches, but also ways to develop more "intelligent" planning strategies can be explored using methods such as GA. A comprehensive study has been put forward that can pave the way for innovative ideas in both academic and industrial terms.

C.) Creativity Part

Hypothetical Term Paper for ACM 369: "Dynamic Priority Aging in CPU Scheduling"

Background and Rationale

In many operating systems, traditional CPU scheduling algorithms (for example, FCFS, SJF, Round Robin, or Prioritization Scheduling) constantly encounter a problem: starvation. For example, a low-priority transaction in the Prioritization Schedule can wait indefinitely if higher-priority transactions arrive constantly. To overcome such problems, operating systems usually apply the aging technique. This technique increases the priority of a pending process over time, allowing it to take processor time.

Definition of Homework

Design and implement a CPU timing simulator that prevents starvation using dynamic priority aging:

a.) Processes and Initial Settings

Define a set of processes with the following attributes:

PID (Process ID)

Arrival Time (Arrival Time)

Processing Time (Burst Time)

Initial Priority (Initial Priority)

Set an appropriate December for the priority levels (for example, 1-10, where 1 represents the highest priority).

b.) Traditional Prioritization Timing

Apply a non-preemptive Prioritization Scheduling algorithm that shows that high-priority processes get the CPU first, and low-priority processes can wait.

c.) Dynamic Priority Aging

Improve your Prioritization Scheduling algorithm using dynamic aging:

For example, the priority of an operation increases with each X unit of waiting time (the priority number decreases, if 1 is the highest priority).

Show that pending transactions have moved to the top of the queue over time.

d.) Visualization and Reporting

Provide at least one Gantt graph that shows how dynamic aging changes the order of transaction execution over time.

Calculate and show these metrics:

Average standby time, average return time, and (optional) CPU utilization rate for Basic Prioritization Timing (without aging).

The same metrics for dynamic aging Prioritization Timing.

e.) Comparison and Analysis

Write a short analysis comparing the two scenarios:

How often were low-priority procedures performed without aging?

How has aging affected average waiting times and fairness?

f.) Possible Extensions

Converting the approach to preemptive scheduling, let it take over the CPU when the priority of a pending process is raised.

Add Round Robin time slots (quantum) within each priority level.

Observe the effect of aging on CPU-oriented processes by integrating I/O-oriented processes.

The Reason for The Choice

a.) Addresses a Real Operating System Problem

Priority-based scheduling is a common method, but it naturally has some disadvantages. This assignment allows students to learn a practical method (aging) to reduce hunger.

b.) Gives Hands-On Experience

Students should design and code a timing simulator, do visualization with Gantt charts, and measure metrics. This mimics the real-world operating system experience.

c.) Encourages Critical Thinking

Students learn to evaluate the balance between performance, complexity and fairness by comparing a Deceleration-based strategy with a standard Prioritization Schedule.

This assignment invites students to add an important feature used in many production systems by developing a classical scheduling algorithm (Prioritization Scheduling). The goal is to develop creative problem solving skills while improving fairness and performance.