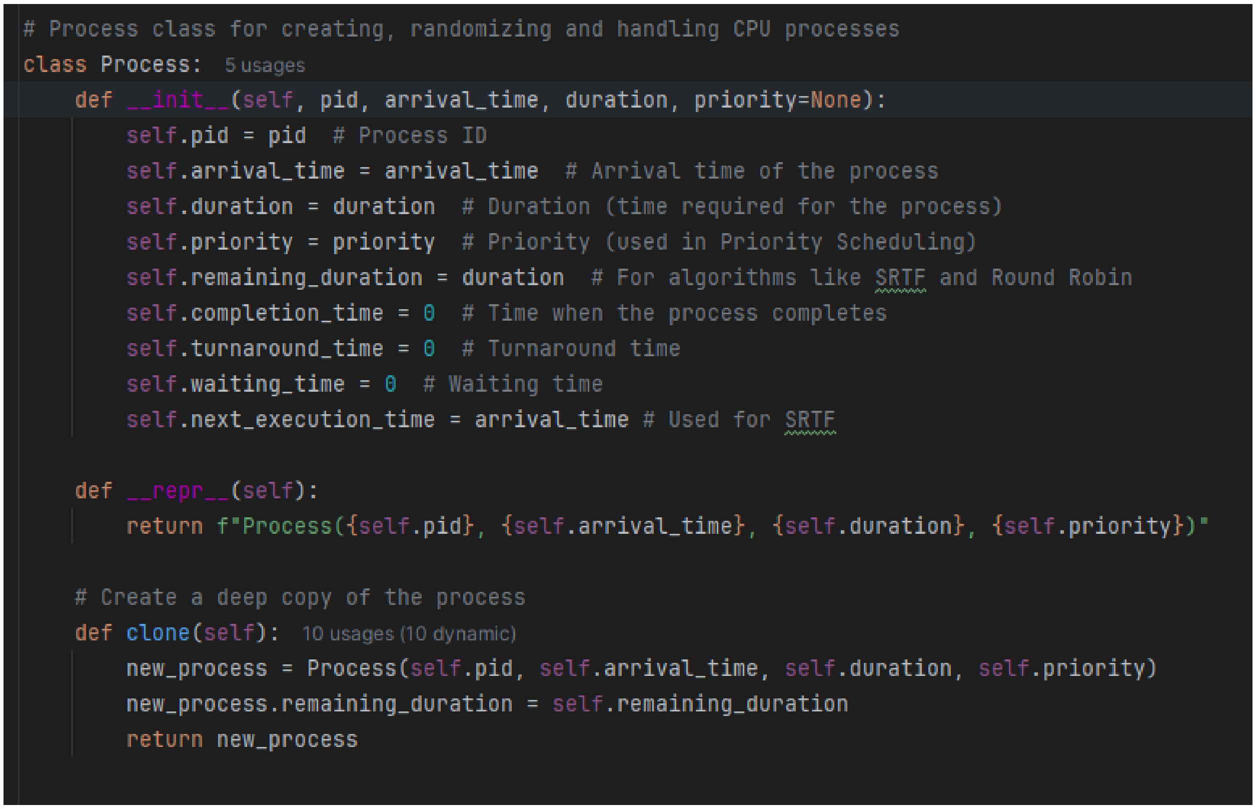
Tom Steinman
3/3/2025

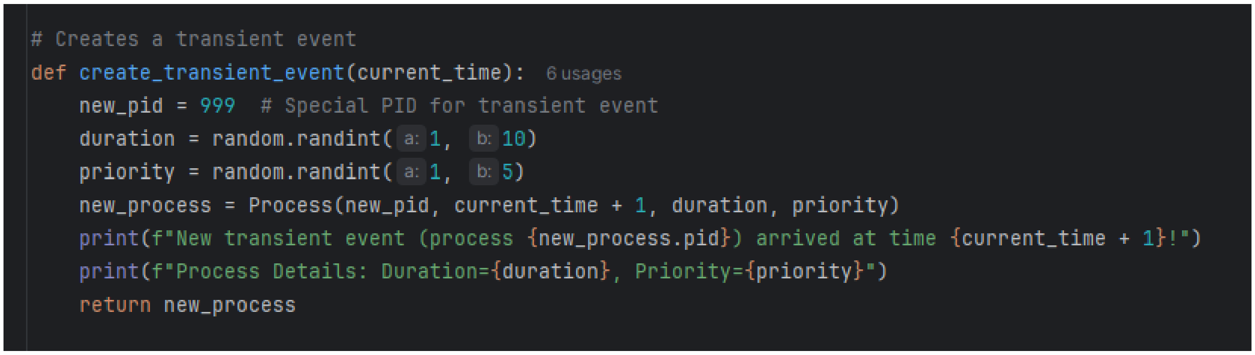
CS5531-0001
Project 1 CPU Scheduling
https://github.com/tasn78/CPU-Scheduling

For this project, I created a Process class for creating and handling CPU process attributes. I also created a clone method to deep copy the process list so that transient events from previous algorithms were not added in each new algorithm handling the processes.
# process.py



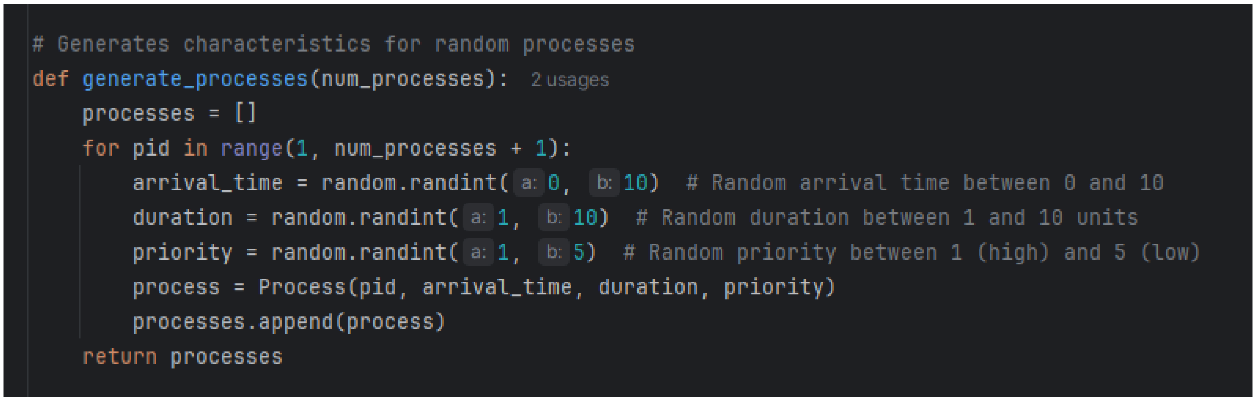
```
# Process class for creating, randomizing and handling CPU processes
class Process:  5 usages
    def __init__(self, pid, arrival_time, duration, priority=None):
        self.pid = pid  # Process ID
        self.arrival_time = arrival_time  # Arrival time of the process
        self.duration = duration  # Duration (time required for the process)
        self.priority = priority  # Priority (used in Priority Scheduling)
        self.remaining_duration = duration  # For algorithms like SRTF and Round Robin
        self.completion_time = 0  # Time when the process completes
        self.turnaround_time = 0  # Turnaround time
        self.waiting_time = 0  # Waiting time
        self.next_execution_time = arrival_time # Used for SRTF

    def __repr__(self):
        return f"Process({self.pid}, {self.arrival_time}, {self.duration}, {self.priority})"

    # Create a deep copy of the process
    def clone(self):  10 usages (10 dynamic)
        new_process = Process(self.pid, self.arrival_time, self.duration, self.priority)
        new_process.remaining_duration = self.remaining_duration
        return new_process
```

I created the create_transient_event method to simulate a newly created transient event to appear randomly in each algorithm. Each transient event is given a process ID of 999 to differentiate between the processes created and the transient event.

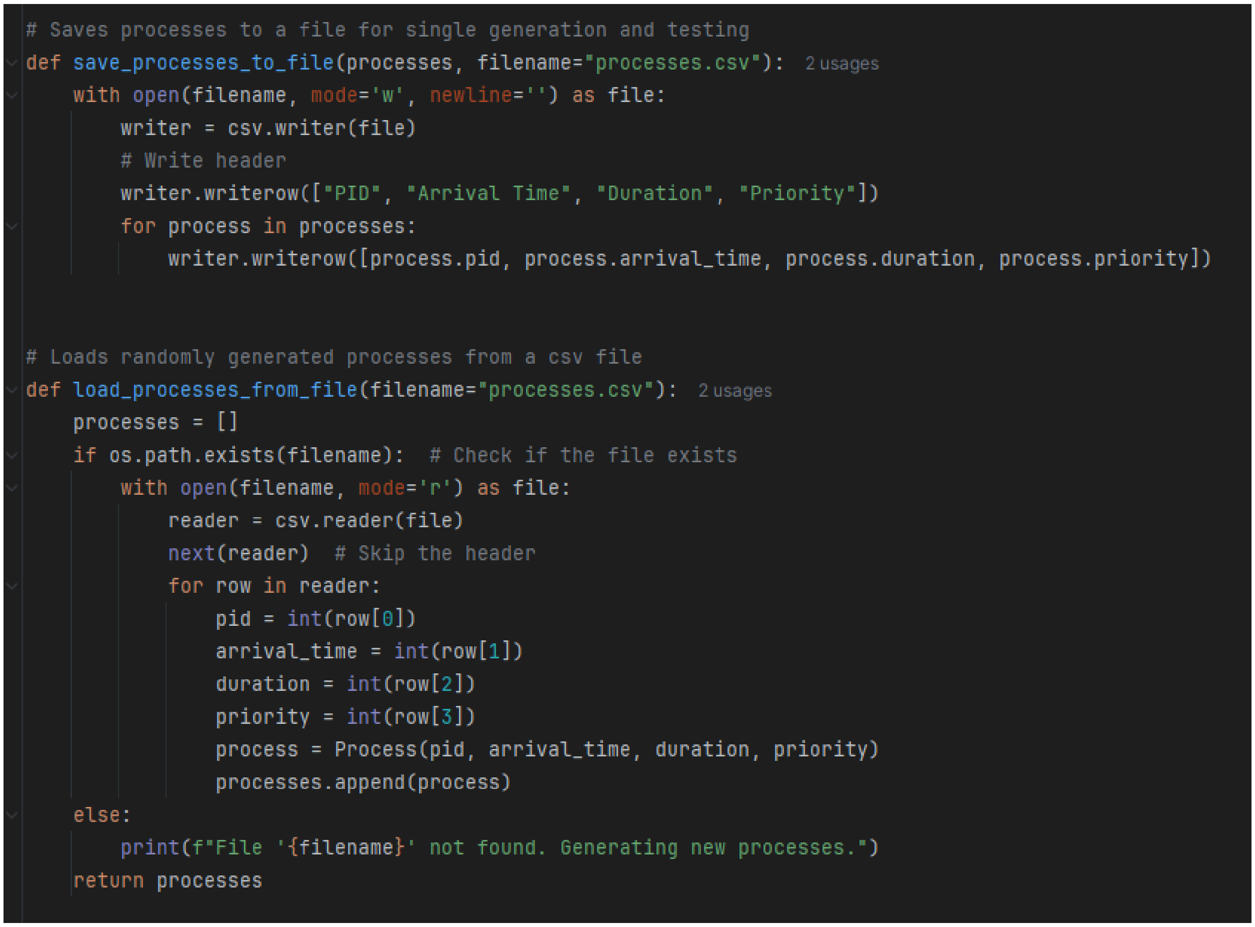


```
# Creates a transient event
def create_transient_event(current_time):  6 usages
    new_pid = 999  # Special PID for transient event
    duration = random.randint( a: 1,  b: 10)
    priority = random.randint( a: 1,  b: 5)
    new_process = Process(new_pid, current_time + 1, duration, priority)
    print(f"New transient event (process {new_process.pid}) arrived at time {current_time + 1}!")
    print(f"Process Details: Duration={duration}, Priority={priority}")
    return new_process
```

For more consistent testing, I randomly created processes using generate_processes, saved the processes to processes.csv with 13 processes (my student ID ends in 03 + 10 = 13) using save_processes_to_file. I then loaded the file to be used in each algorithm using the method load_processes_from_file. These are both commented out in main.py, but can be used to generate and load new random processes.



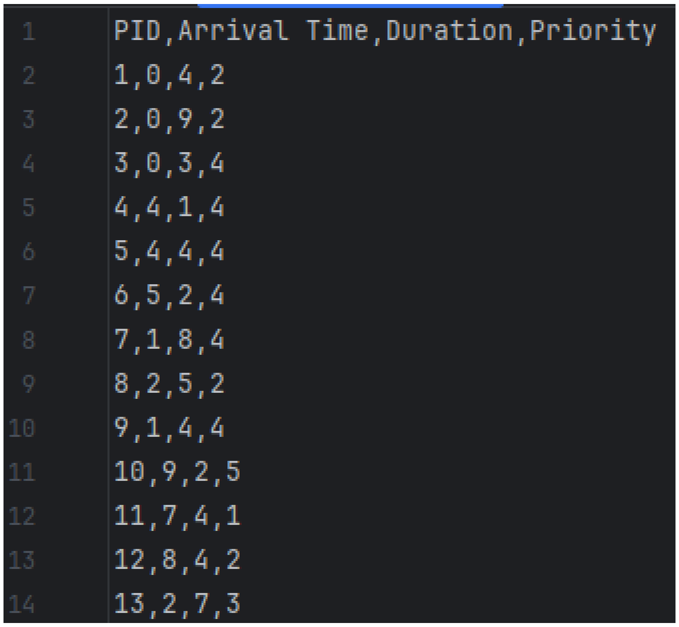
```python
# Generates characteristics for random processes
def generate_processes(num_processes):  2 usages
    processes = []
    for pid in range(1, num_processes + 1):
        arrival_time = random.randint( a: 0,  b: 10)  # Random arrival time between 0 and 10
        duration = random.randint( a: 1,  b: 10)  # Random duration between 1 and 10 units
        priority = random.randint( a: 1,  b: 5)  # Random priority between 1 (high) and 5 (low)
        process = Process(pid, arrival_time, duration, priority)
        processes.append(process)
    return processes
```



```python
# Saves processes to a file for single generation and testing
def save_processes_to_file(processes, filename="processes.csv"):  2 usages
    with open(filename, mode='w', newline='') as file:
        writer = csv.writer(file)
        # Write header
        writer.writerow(["PID", "Arrival Time", "Duration", "Priority"])
        for process in processes:
            writer.writerow([process.pid, process.arrival_time, process.duration, process.priority])


# Loads randomly generated processes from a csv file
def load_processes_from_file(filename="processes.csv"):  2 usages
    processes = []
    if os.path.exists(filename):  # Check if the file exists
        with open(filename, mode='r') as file:
            reader = csv.reader(file)
            next(reader)  # Skip the header
            for row in reader:
                pid = int(row[0])
                arrival_time = int(row[1])
                duration = int(row[2])
                priority = int(row[3])
                process = Process(pid, arrival_time, duration, priority)
                processes.append(process)
    else:
        print(f"File '{filename}' not found. Generating new processes.")
    return processes
```

Tom Steinman
3/3/2025

# processes.csv
This is the file used in output below:

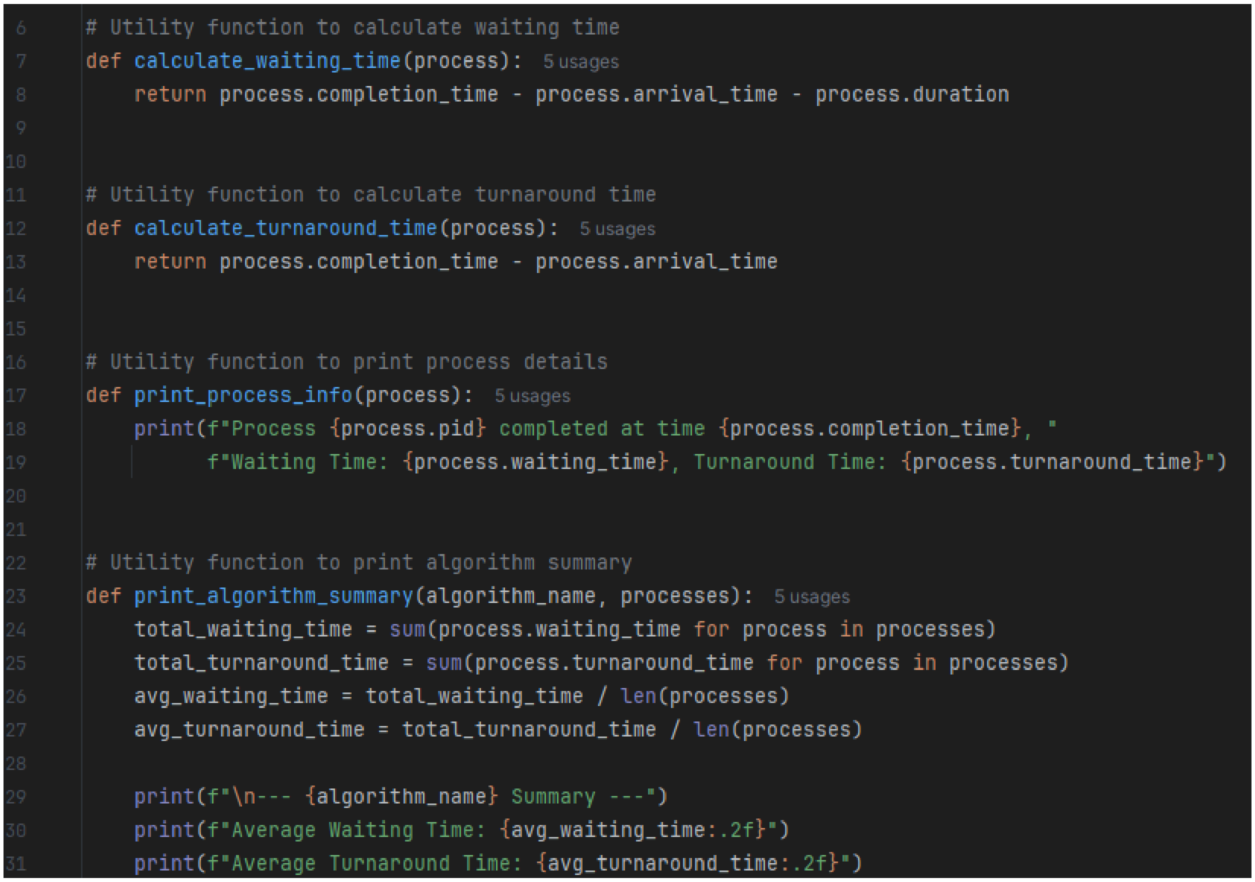


```
PID,Arrival Time,Duration,Priority
1,0,4,2
2,0,9,2
3,0,3,4
4,4,1,4
5,4,4,4
6,5,2,4
7,1,8,4
8,2,5,2
9,1,4,4
10,9,2,5
11,7,4,1
12,8,4,2
13,2,7,3
```

# scheduling.py

I created utility functions to calculate waiting time, turnaround time, printing and an algorithm summary to calculate and show the performances of each algorithm.

Tom Steinman
3/3/2025



```python
# Utility function to calculate waiting time
def calculate_waiting_time(process):
    return process.completion_time - process.arrival_time - process.duration


# Utility function to calculate turnaround time
def calculate_turnaround_time(process):
    return process.completion_time - process.arrival_time


# Utility function to print process details
def print_process_info(process):
    print(f"Process {process.pid} completed at time {process.completion_time}, "
          f"Waiting Time: {process.waiting_time}, Turnaround Time: {process.turnaround_time}")


# Utility function to print algorithm summary
def print_algorithm_summary(algorithm_name, processes):
    total_waiting_time = sum(process.waiting_time for process in processes)
    total_turnaround_time = sum(process.turnaround_time for process in processes)
    avg_waiting_time = total_waiting_time / len(processes)
    avg_turnaround_time = total_turnaround_time / len(processes)

    print(f"\n--- {algorithm_name} Summary ---")
    print(f"Average Waiting Time: {avg_waiting_time:.2f}")
    print(f"Average Turnaround Time: {avg_turnaround_time:.2f}")
```

The 5 algorithms I chose were FCFS, SJF, Priority Scheduling, Shortest Time Remaining First and Round Robin. In every algorithm I create a deep copy so that the random transient event is not added to the list for preceding algorithms. I set the event time for the transient event to take place between 5-15 (event_time) in every algorithm so that the event always appears in the middle of an algorithm to assure it is working properly. Each algorithm prints the start and completion of each process, when the transient event arrives, the transient event details, when it completes and the summary.

FCFS

The first come first served algorithm uses the arrival time to determine when a process is completed. In this case, if processes arrive at the same time, they are completed in the order of the list that they appear after being sorted by arrival time. Using a for loop, the processes are completed based on arrival time, with the first if statement being used if the transient event has not been created yet using event_time between 5-15. Once the transient event is created, the next if statement (if transient_process) will ensure that the transient event is completed before moving on to the next process that arrived. The details of each process and the summary will be printed.

```python
# FCFS (First Come First Served) Scheduling
def fcfs_scheduling(original_processes):
   # Create a copy of processes to avoid modifying the original list
   processes = [process.clone() for process in original_processes]
```

Tom Steinman
3/3/2025

```
    processes.sort(key=lambda x: x.arrival_time)  # Sort by arrival time

    current_time = 0
    event_time = random.randint(5, 15)  # Set a single event time for this run
    transient_event_triggered = False
    transient_process = None
    completed_processes = []

    i = 0
    while i < len(processes):
        # Check if the transient event should be triggered (only once)
        if not transient_event_triggered and current_time >= event_time:
            # Create the transient event
            transient_process = create_transient_event(current_time)

            # Set the arrival time to the current time
            transient_process.arrival_time = current_time

            # Immediately process the transient event before continuing
            if current_time < transient_process.arrival_time:
                current_time = transient_process.arrival_time

            print(f"Starting Transient Process {transient_process.pid} at time {current_time}")
            transient_process.completion_time = current_time + transient_process.duration
            transient_process.turnaround_time = calculate_turnaround_time(transient_process)
            transient_process.waiting_time = calculate_waiting_time(transient_process)

            # Update current time
            current_time = transient_process.completion_time
            print_process_info(transient_process)
            completed_processes.append(transient_process)

            # Mark the transient event as triggered
            transient_event_triggered = True

        # Handle regular processes
        process = processes[i]
        # Ensure process starts when it arrives
        if current_time < process.arrival_time:
            current_time = process.arrival_time

        print(f"Starting Process {process.pid} at time {current_time}")

        # Calculate completion time
        process.completion_time = current_time + process.duration
        process.turnaround_time = calculate_turnaround_time(process)
        process.waiting_time = calculate_waiting_time(process)

        # Update current time for the next process
        current_time = process.completion_time

        # Print process info
        print_process_info(process)
        completed_processes.append(process)
        i += 1

    print_algorithm_summary("FCFS", completed_processes)
    return completed_processes
```

SJF

Due to the various arrival times, I created a counter (current_time) to keep track of time and determine the processes that arrived. Based on the remaining_processes (processes that have arrived) they are sorted by arrival time in that the available_processes are only the processes that have “arrived” so that the process arrival time must be less than or equal to the current_time. The while loop continues completing the shortest job first. Each process start and completion time will be printed, as well as the time the transient event arrives and when it is completed. The first if statement will move the current time to the next available process if none are available. The next if statement will process the transient event if it has arrived. Otherwise, it will complete the process list according to the shortest job first.

```
# SJF (Shortest Job First) Scheduling
def sjf_scheduling(original_processes):
    # Create a copy of processes to avoid modifying the original list
    processes = [process.clone() for process in original_processes]

    current_time = 0
    event_time = random.randint(5, 15)  # Set a single event time for this run
    transient_event_triggered = False
    remaining_processes = sorted(processes, key=lambda x: x.arrival_time)
    completed_processes = []

    while remaining_processes:
        # Check if the transient event should be triggered (only once)
        if not transient_event_triggered and current_time >= event_time:
            # Create the transient event
            transient_process = create_transient_event(current_time)

            # Set the arrival time to the current time
            transient_process.arrival_time = current_time

            # Immediately process the transient event before continuing
```

Tom Steinman
3/3/2025

```
            if current_time < transient_process.arrival_time:
                current_time = transient_process.arrival_time

            print(f"Starting Transient Process {transient_process.pid} at time
{current_time}")
            transient_process.completion_time = current_time +
transient_process.duration
            transient_process.turnaround_time =
calculate_turnaround_time(transient_process)
            transient_process.waiting_time =
calculate_waiting_time(transient_process)

            # Update current time
            current_time = transient_process.completion_time
            print_process_info(transient_process)
            completed_processes.append(transient_process)

            # Mark the transient event as triggered
            transient_event_triggered = True

        # Get available processes at the current time
        available_processes = [p for p in remaining_processes if p.arrival_time
<= current_time]

        # No available processes - jump to next arrival time
        if not available_processes:
            current_time = min(p.arrival_time for p in remaining_processes)
            continue

        # Select process with the shortest duration among the available
processes
        next_process = min(available_processes, key=lambda x: x.duration)
        print(f"Starting Process {next_process.pid} at time {current_time}")

        # Calculate completion time for the selected process
        next_process.completion_time = current_time + next_process.duration
        next_process.turnaround_time = calculate_turnaround_time(next_process)
        next_process.waiting_time = calculate_waiting_time(next_process)

        # Update current time
        current_time = next_process.completion_time

        # Print process info
        print_process_info(next_process)

        # Move process from remaining to completed
        remaining_processes.remove(next_process)
        completed_processes.append(next_process)
```

Tom Steinman
3/3/2025

```
    print_algorithm_summary("SJF", completed_processes)
    return completed_processes
```

Priority Scheduling

Priority scheduling also uses a while loop, but in this case it is based on the priority attribute of each process. The processes will again be based on arrival time, however if a new process arrives with a higher priority than it will be completed before processes with a lower priority. In this case, the highest priority is 1 with the lowest priority being 5 with priority processes being determined by `next_process = min(available_processes, key=lambda x: x.priority)`. The if statement if transient_process, allows the transient event to be processed as soon as it arrives before all other processes. Afterwards, the remaining processes will be completed based on arrival time, as usual, with the highest priority processes first.

```
# Priority Scheduling
def priority_scheduling(original_processes):
    # Create a copy of processes to avoid modifying the original list
    processes = [process.clone() for process in original_processes]

    current_time = 0
    event_time = random.randint(5, 15)  # Set a single event time for this run
    transient_event_triggered = False
    remaining_processes = sorted(processes, key=lambda x: x.arrival_time)
    completed_processes = []

    while remaining_processes:
        # Check if the transient event should be triggered (only once)
        if not transient_event_triggered and current_time >= event_time:
            # Create the transient event
            transient_process = create_transient_event(current_time)

            # Set the arrival time to the current time
            transient_process.arrival_time = current_time

            # Immediately process the transient event before continuing
            if current_time < transient_process.arrival_time:
                current_time = transient_process.arrival_time

            print(
                f"Starting Transient Process {transient_process.pid} (Priority {transient_process.priority}) at time {current_time}")
            transient_process.completion_time = current_time + transient_process.duration
            transient_process.turnaround_time = calculate_turnaround_time(transient_process)
            transient_process.waiting_time = calculate_waiting_time(transient_process)
            # Update current time
            current_time = transient_process.completion_time
            print_process_info(transient_process)
            completed_processes.append(transient_process)

            # Mark the transient event as triggered
            transient_event_triggered = True

        # Get available processes at current time
        available_processes = [p for p in remaining_processes if p.arrival_time <= current_time]

        if not available_processes:
            # Jump to the next arrival time if no processes are available
            current_time = min(p.arrival_time for p in remaining_processes)
            continue

        # Select process with the highest priority (lowest number) among the available processes
        next_process = min(available_processes, key=lambda x: x.priority)

        print(f"Starting Process {next_process.pid} (Priority {next_process.priority}) at time {current_time}")

        # Calculate completion time
        next_process.completion_time = current_time + next_process.duration
        next_process.turnaround_time = calculate_turnaround_time(next_process)
        next_process.waiting_time = calculate_waiting_time(next_process)

        # Update current time
        current_time = next_process.completion_time

        # Print process info
        print_process_info(next_process)

        # Move process from remaining to completed
        remaining_processes.remove(next_process)
        completed_processes.append(next_process)

    print_algorithm_summary("Priority Scheduling", completed_processes)
    return completed_processes
```

SRTF (Shortest Remaining Time First)

The shortest remaining time first algorithm uses similar qualities as the last algorithm, using arrival time to determine if there are processes in the list, creating a transient event between 5-15 of the current_time, and processing the transient event first as it arrives before other processes. The main difference in this code, is that it completes the processes based on remaining_duration of the processes in the list, and the key feature of SRTF is that it can preempt processes if a new one arrives with a shorter remaining time. This is handled in this line of code:

```
# Determine how long this process will run
        if next_arrival_time != float('inf') and next_arrival_time <
current_time + next_process.remaining_duration:
            # Process will be preempted by a new arrival
```

In this case, the shortest remaining time of each process in the list of arrived processes is completed first. The process is completed and the details are calculated and printed. When all processes are complete, the summary is printed as well.

```
# SRTF (Shortest Remaining Time First) Scheduling
def srtf_scheduling(original_processes):
    # Create a copy of processes to avoid modifying the original list
    processes = [process.clone() for process in original_processes]

    print("\n--- SRTF (Shortest Remaining Time First) Scheduling ---")

    current_time = 0
    event_time = random.randint(5, 15)  # Set a single event time for this run
    transient_event_triggered = False
    remaining_processes = sorted(processes, key=lambda x: x.arrival_time)
    completed_processes = []

    while remaining_processes:
        # Check if the transient event should be triggered (only once)
        if not transient_event_triggered and current_time >= event_time:
            # Create the transient event
            transient_process = create_transient_event(current_time)

            # Set the arrival time to the current time
            transient_process.arrival_time = current_time

            # Immediately process the transient event before continuing
            if current_time < transient_process.arrival_time:
                current_time = transient_process.arrival_time

            print(
                f"Starting Transient Process {transient_process.pid} (Remaining:
{transient_process.remaining_duration}) at time {current_time}")
            transient_process.completion_time = current_time +
transient_process.duration
            transient_process.turnaround_time =
calculate_turnaround_time(transient_process)
            transient_process.waiting_time =
calculate_waiting_time(transient_process)

            # Update current time
            current_time = transient_process.completion_time
            print_process_info(transient_process)
            completed_processes.append(transient_process)

            # Mark the transient event as triggered
            transient_event_triggered = True

        # Get available processes at current time
        available_processes = [p for p in remaining_processes if p.arrival_time
<= current_time]

        if not available_processes:
            # Jump to the next arrival time if no processes are available
            current_time = min(p.arrival_time for p in remaining_processes)
            continue

        # Select process with shortest remaining time from the available
processes
        next_process = min(available_processes, key=lambda x:
x.remaining_duration)

        print(
            f"Starting/Resuming Process {next_process.pid} (Remaining:
{next_process.remaining_duration}) at time {current_time}")

        # Determine the time until the next process arrival
        next_arrival_time = float('inf')
        for p in remaining_processes:
            if p.arrival_time > current_time:
                next_arrival_time = min(next_arrival_time, p.arrival_time)

        # Determine how long this process will run
        if next_arrival_time != float('inf') and next_arrival_time <
current_time + next_process.remaining_duration:
            # Process will be preempted by a new arrival
            time_slice = next_arrival_time - current_time
            next_process.remaining_duration -= time_slice
            current_time = next_arrival_time
            print(
                f"Process {next_process.pid} preempted at time {current_time},
remaining: {next_process.remaining_duration}")
        else:
            # Process will complete
            current_time += next_process.remaining_duration
            next_process.remaining_duration = 0
            next_process.completion_time = current_time
            next_process.turnaround_time =
calculate_turnaround_time(next_process)
            next_process.waiting_time = calculate_waiting_time(next_process)
```

Tom Steinman
3/3/2025

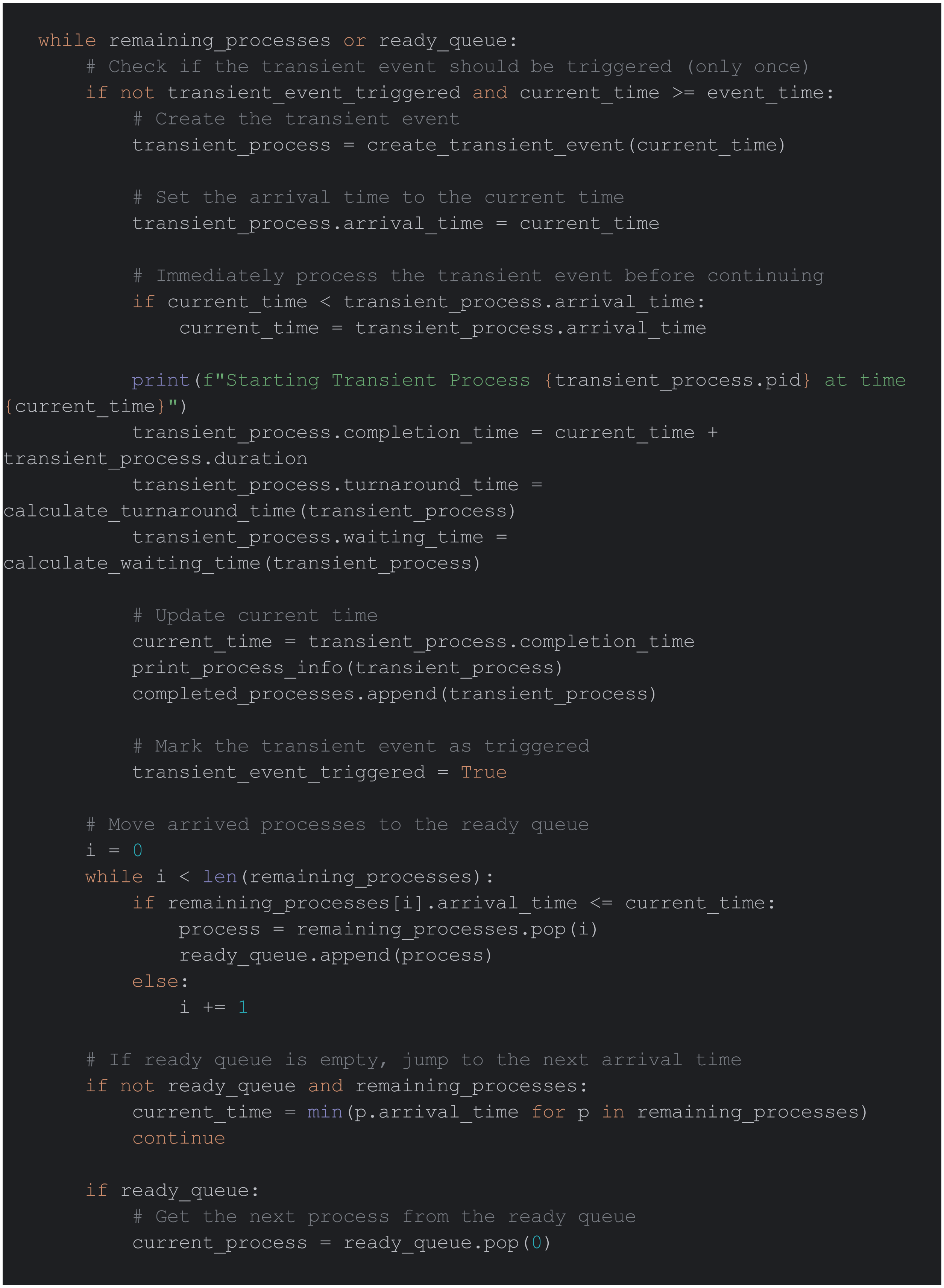
```
            # Print process info
            print_process_info(next_process)

            # Move process from remaining to completed
            remaining_processes.remove(next_process)
            completed_processes.append(next_process)

    print_algorithm_summary("SRTF", completed_processes)
    return completed_processes
```

Round Robin

The round robin algorithm differs from the other algorithms coded, in that it uses a queue for processes that are ready. As the process “arrives” from the list, it is popped and appended to the queue(ready_queue). It also uses a time quantum, determined in main.py (I used a time quantum of 2). This uses 2 while loops, the outer one continuing while any processes are in the list or queue, with the inner while loop continuing until the remaining_process list is emptied. If no process has arrived, it updates to the next arrival time. The transient event is handled similarly to the other algorithms using if statements. As long as one hasn’t arrived it uses a time quantum of 2 to complete the processes in increments of time 2 to all of the processes that arrived. As new processes arrive, it includes them in a circular manner, completing 2 time durations to each process until it is complete, removed from the queue and prints the results. In this case, I also used an if statement (if transient_process) to complete the transient process first before all other processes using the time quantum (time_quantum) methodology of round robin scheduling. In order to deal with processes potentially finishing during the time quantum, I used time_slice to keep track of the time_quantum details that would factor in the completion times.

```
# Round Robin Scheduling
def round_robin_scheduling(original_processes, time_quantum):
    # Create a copy of processes to avoid modifying the original list
    processes = [process.clone() for process in original_processes]

    print(f"\n--- Round Robin Scheduling (Time Quantum = {time_quantum}) ---")

    current_time = 0
    event_time = random.randint(5, 15)  # Set a single event time for this run
    transient_event_triggered = False

    # Sort processes by arrival time
    processes.sort(key=lambda x: x.arrival_time)

    # Create a ready queue
    ready_queue = []
    completed_processes = []
    remaining_processes = processes.copy()
```

Tom Steinman
3/3/2025



```
    while remaining_processes or ready_queue:
        # Check if the transient event should be triggered (only once)
        if not transient_event_triggered and current_time >= event_time:
            # Create the transient event
            transient_process = create_transient_event(current_time)

            # Set the arrival time to the current time
            transient_process.arrival_time = current_time

            # Immediately process the transient event before continuing
            if current_time < transient_process.arrival_time:
                current_time = transient_process.arrival_time

            print(f"Starting Transient Process {transient_process.pid} at time
{current_time}")
            transient_process.completion_time = current_time +
transient_process.duration
            transient_process.turnaround_time =
calculate_turnaround_time(transient_process)
            transient_process.waiting_time =
calculate_waiting_time(transient_process)

            # Update current time
            current_time = transient_process.completion_time
            print_process_info(transient_process)
            completed_processes.append(transient_process)

            # Mark the transient event as triggered
            transient_event_triggered = True

        # Move arrived processes to the ready queue
        i = 0
        while i < len(remaining_processes):
            if remaining_processes[i].arrival_time <= current_time:
                process = remaining_processes.pop(i)
                ready_queue.append(process)
            else:
                i += 1

        # If ready queue is empty, jump to the next arrival time
        if not ready_queue and remaining_processes:
            current_time = min(p.arrival_time for p in remaining_processes)
            continue

        if ready_queue:
            # Get the next process from the ready queue
            current_process = ready_queue.pop(0)
```

Tom Steinman
3/3/2025

```
            print(
                f"Starting/Resuming Process {current_process.pid} at time {current_time} (Remaining: {current_process.remaining_duration})")

            # Normal process scheduling using time quantum
            if current_process.remaining_duration <= time_quantum:
                # Process will complete within this time quantum
                time_slice = current_process.remaining_duration
                current_process.remaining_duration = 0
                current_process.completion_time = current_time + time_slice
                current_process.turnaround_time = calculate_turnaround_time(current_process)
                current_process.waiting_time = calculate_waiting_time(current_process)

                # Print process info
                print_process_info(current_process)

                # Add to completed processes
                completed_processes.append(current_process)
            else:
                # Process will use the full time quantum
                time_slice = time_quantum
                current_process.remaining_duration -= time_quantum

                print(
                    f"Process {current_process.pid} used its time quantum, remaining: {current_process.remaining_duration}")

                # Update current time before adding back to ready queue
                current_time += time_slice

                # Add back to ready queue
                ready_queue.append(current_process)
                continue  # Skip the time update at the end since we already updated it

            # Update the current time
            current_time += time_slice

    print_algorithm_summary("Round Robin", completed_processes)
    return completed_processes
```

# main.py

```
# main.py

import os
```

Tom Steinman
3/3/2025

```python
from process import Process, generate_processes, save_processes_to_file,
load_processes_from_file
from scheduling import fcfs_scheduling, sjf_scheduling, priority_scheduling,
srtf_scheduling, round_robin_scheduling


def main():
    # Define the file where processes are saved
    filename = "processes.csv"

    # Check if the file exists
    if os.path.exists(filename):
        # Load processes from the file
        original_processes = load_processes_from_file(filename)
        print("Loaded processes from file:")
    else:
        # Generate 13 processes (last two digits of your ID + 10 if needed)
        original_processes = generate_processes(13)
        # Save the generated processes to a file for future use
        save_processes_to_file(original_processes, filename)
        print("Generated new processes and saved to file:")

    # Print out the loaded/generated processes
    for process in original_processes:
        print(process)

    # First Come First Serve (FCFS)
    print("\n--- FCFS Scheduling ---")
    fcfs_results = fcfs_scheduling(original_processes)

    # Shortest Job First (SJF)
    print("\n--- SJF Scheduling ---")
    sjf_results = sjf_scheduling(original_processes)

    # Priority Scheduling
    print("\n--- Priority Scheduling ---")
    priority_results = priority_scheduling(original_processes)

    # Shortest Remaining Time First (SRTF)
    print("\n--- SRTF Scheduling ---")
    srtf_results = srtf_scheduling(original_processes)

    # Round Robin Scheduling (with a time quantum of 2 units)
    print("\n--- Round Robin Scheduling ---")
    rr_results = round_robin_scheduling(original_processes, time_quantum=2)

    # Compare all algorithms
    # Calculate average metrics for each algorithm
```

```
   avg_waiting_fcfs = sum(p.waiting_time for p in fcfs_results) /
len(fcfs_results)
   avg_turnaround_fcfs = sum(p.turnaround_time for p in fcfs_results) /
len(fcfs_results)

   avg_waiting_sjf = sum(p.waiting_time for p in sjf_results) /
len(sjf_results)
   avg_turnaround_sjf = sum(p.turnaround_time for p in sjf_results) /
len(sjf_results)

   avg_waiting_priority = sum(p.waiting_time for p in priority_results) /
len(priority_results)
   avg_turnaround_priority = sum(p.turnaround_time for p in priority_results) /
len(priority_results)

   avg_waiting_srtf = sum(p.waiting_time for p in srtf_results) /
len(srtf_results)
   avg_turnaround_srtf = sum(p.turnaround_time for p in srtf_results) /
len(srtf_results)

   avg_waiting_rr = sum(p.waiting_time for p in rr_results) / len(rr_results)
   avg_turnaround_rr = sum(p.turnaround_time for p in rr_results) /
len(rr_results)

   print("\n--- Comparison of Scheduling Algorithms ---")
   print("{:<15} {:<20} {:<20}".format("Algorithm", "Avg Waiting Time", "Avg
Turnaround Time"))
   print("-" * 60)
   print("{:<15} {:<20.2f} {:<20.2f}".format("FCFS", avg_waiting_fcfs,
avg_turnaround_fcfs))
   print("{:<15} {:<20.2f} {:<20.2f}".format("SJF", avg_waiting_sjf,
avg_turnaround_sjf))
   print("{:<15} {:<20.2f} {:<20.2f}".format("Priority", avg_waiting_priority,
avg_turnaround_priority))
   print("{:<15} {:<20.2f} {:<20.2f}".format("SRTF", avg_waiting_srtf,
avg_turnaround_srtf))
   print("{:<15} {:<20.2f} {:<20.2f}".format("Round Robin", avg_waiting_rr,
avg_turnaround_rr))


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

Output

Loaded processes from file:
Process(1, 0, 4, 2)

Tom Steinman
3/3/2025

```
Process(2, 0, 9, 2)
Process(3, 0, 3, 4)
Process(4, 4, 1, 4)
Process(5, 4, 4, 4)
Process(6, 5, 2, 4)
Process(7, 1, 8, 4)
Process(8, 2, 5, 2)
Process(9, 1, 4, 4)
Process(10, 9, 2, 5)
Process(11, 7, 4, 1)
Process(12, 8, 4, 2)
Process(13, 2, 7, 3)

--- FCFS Scheduling ---
Starting Process 1 at time 0
Process 1 completed at time 4, Waiting Time: 0, Turnaround Time: 4
Starting Process 2 at time 4
Process 2 completed at time 13, Waiting Time: 4, Turnaround Time: 13
Starting Process 3 at time 13
Process 3 completed at time 16, Waiting Time: 13, Turnaround Time: 16
New transient event (process 999) arrived at time 17!
Process Details: Duration=10, Priority=4
Starting Transient Process 999 at time 16
Process 999 completed at time 26, Waiting Time: 0, Turnaround Time: 10
Starting Process 7 at time 26
Process 7 completed at time 34, Waiting Time: 25, Turnaround Time: 33
Starting Process 9 at time 34
Process 9 completed at time 38, Waiting Time: 33, Turnaround Time: 37
Starting Process 8 at time 38
Process 8 completed at time 43, Waiting Time: 36, Turnaround Time: 41
Starting Process 13 at time 43
Process 13 completed at time 50, Waiting Time: 41, Turnaround Time: 48
Starting Process 4 at time 50
Process 4 completed at time 51, Waiting Time: 46, Turnaround Time: 47
Starting Process 5 at time 51
Process 5 completed at time 55, Waiting Time: 47, Turnaround Time: 51
Starting Process 6 at time 55
Process 6 completed at time 57, Waiting Time: 50, Turnaround Time: 52
Starting Process 11 at time 57
Process 11 completed at time 61, Waiting Time: 50, Turnaround Time: 54
Starting Process 12 at time 61
Process 12 completed at time 65, Waiting Time: 53, Turnaround Time: 57
Starting Process 10 at time 65
Process 10 completed at time 67, Waiting Time: 56, Turnaround Time: 58
```

Tom Steinman
3/3/2025

```
--- FCFS Summary ---
Average Waiting Time: 32.43
Average Turnaround Time: 37.21

--- SJF Scheduling ---
Starting Process 3 at time 0
Process 3 completed at time 3, Waiting Time: 0, Turnaround Time: 3
Starting Process 1 at time 3
Process 1 completed at time 7, Waiting Time: 3, Turnaround Time: 7
Starting Process 4 at time 7
Process 4 completed at time 8, Waiting Time: 3, Turnaround Time: 4
New transient event (process 999) arrived at time 9!
Process Details: Duration=1, Priority=3
Starting Transient Process 999 at time 8
Process 999 completed at time 9, Waiting Time: 0, Turnaround Time: 1
Starting Process 6 at time 9
Process 6 completed at time 11, Waiting Time: 4, Turnaround Time: 6
Starting Process 10 at time 11
Process 10 completed at time 13, Waiting Time: 2, Turnaround Time: 4
Starting Process 9 at time 13
Process 9 completed at time 17, Waiting Time: 12, Turnaround Time: 16
Starting Process 5 at time 17
Process 5 completed at time 21, Waiting Time: 13, Turnaround Time: 17
Starting Process 11 at time 21
Process 11 completed at time 25, Waiting Time: 14, Turnaround Time: 18
Starting Process 12 at time 25
Process 12 completed at time 29, Waiting Time: 17, Turnaround Time: 21
Starting Process 8 at time 29
Process 8 completed at time 34, Waiting Time: 27, Turnaround Time: 32
Starting Process 13 at time 34
Process 13 completed at time 41, Waiting Time: 32, Turnaround Time: 39
Starting Process 7 at time 41
Process 7 completed at time 49, Waiting Time: 40, Turnaround Time: 48
Starting Process 2 at time 49
Process 2 completed at time 58, Waiting Time: 49, Turnaround Time: 58

--- SJF Summary ---
Average Waiting Time: 15.43
Average Turnaround Time: 19.57

--- Priority Scheduling ---
Starting Process 1 (Priority 2) at time 0
Process 1 completed at time 4, Waiting Time: 0, Turnaround Time: 4
```

Tom Steinman
3/3/2025

```
Starting Process 2 (Priority 2) at time 4
Process 2 completed at time 13, Waiting Time: 4, Turnaround Time: 13
Starting Process 11 (Priority 1) at time 13
Process 11 completed at time 17, Waiting Time: 6, Turnaround Time: 10
New transient event (process 999) arrived at time 18!
Process Details: Duration=2, Priority=4
Starting Transient Process 999 (Priority 4) at time 17
Process 999 completed at time 19, Waiting Time: 0, Turnaround Time: 2
Starting Process 8 (Priority 2) at time 19
Process 8 completed at time 24, Waiting Time: 17, Turnaround Time: 22
Starting Process 12 (Priority 2) at time 24
Process 12 completed at time 28, Waiting Time: 16, Turnaround Time: 20
Starting Process 13 (Priority 3) at time 28
Process 13 completed at time 35, Waiting Time: 26, Turnaround Time: 33
Starting Process 3 (Priority 4) at time 35
Process 3 completed at time 38, Waiting Time: 35, Turnaround Time: 38
Starting Process 7 (Priority 4) at time 38
Process 7 completed at time 46, Waiting Time: 37, Turnaround Time: 45
Starting Process 9 (Priority 4) at time 46
Process 9 completed at time 50, Waiting Time: 45, Turnaround Time: 49
Starting Process 4 (Priority 4) at time 50
Process 4 completed at time 51, Waiting Time: 46, Turnaround Time: 47
Starting Process 5 (Priority 4) at time 51
Process 5 completed at time 55, Waiting Time: 47, Turnaround Time: 51
Starting Process 6 (Priority 4) at time 55
Process 6 completed at time 57, Waiting Time: 50, Turnaround Time: 52
Starting Process 10 (Priority 5) at time 57
Process 10 completed at time 59, Waiting Time: 48, Turnaround Time: 50

--- Priority Scheduling Summary ---
Average Waiting Time: 26.93
Average Turnaround Time: 31.14

--- SRTF Scheduling ---

--- SRTF (Shortest Remaining Time First) Scheduling ---
Starting/Resuming Process 3 (Remaining: 3) at time 0
Process 3 preempted at time 1, remaining: 2
Starting/Resuming Process 3 (Remaining: 2) at time 1
Process 3 preempted at time 2, remaining: 1
Starting/Resuming Process 3 (Remaining: 1) at time 2
Process 3 completed at time 3, Waiting Time: 0, Turnaround Time: 3
Starting/Resuming Process 1 (Remaining: 4) at time 3
Process 1 preempted at time 4, remaining: 3
```

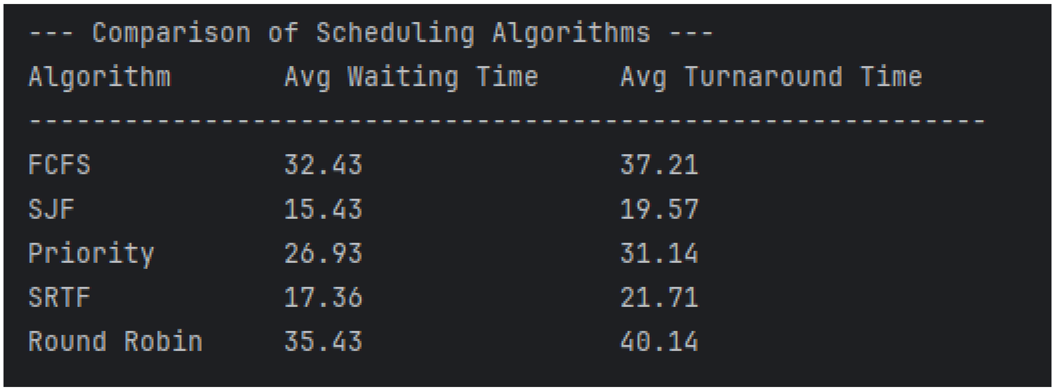
Tom Steinman
3/3/2025

```
Starting/Resuming Process 4 (Remaining: 1) at time 4
Process 4 completed at time 5, Waiting Time: 0, Turnaround Time: 1
Starting/Resuming Process 6 (Remaining: 2) at time 5
Process 6 completed at time 7, Waiting Time: 0, Turnaround Time: 2
Starting/Resuming Process 1 (Remaining: 3) at time 7
Process 1 preempted at time 8, remaining: 2
New transient event (process 999) arrived at time 9!
Process Details: Duration=4, Priority=4
Starting Transient Process 999 (Remaining: 4) at time 8
Process 999 completed at time 12, Waiting Time: 0, Turnaround Time: 4
Starting/Resuming Process 1 (Remaining: 2) at time 12
Process 1 completed at time 14, Waiting Time: 10, Turnaround Time: 14
Starting/Resuming Process 10 (Remaining: 2) at time 14
Process 10 completed at time 16, Waiting Time: 5, Turnaround Time: 7
Starting/Resuming Process 9 (Remaining: 4) at time 16
Process 9 completed at time 20, Waiting Time: 15, Turnaround Time: 19
Starting/Resuming Process 5 (Remaining: 4) at time 20
Process 5 completed at time 24, Waiting Time: 16, Turnaround Time: 20
Starting/Resuming Process 11 (Remaining: 4) at time 24
Process 11 completed at time 28, Waiting Time: 17, Turnaround Time: 21
Starting/Resuming Process 12 (Remaining: 4) at time 28
Process 12 completed at time 32, Waiting Time: 20, Turnaround Time: 24
Starting/Resuming Process 8 (Remaining: 5) at time 32
Process 8 completed at time 37, Waiting Time: 30, Turnaround Time: 35
Starting/Resuming Process 13 (Remaining: 7) at time 37
Process 13 completed at time 44, Waiting Time: 35, Turnaround Time: 42
Starting/Resuming Process 7 (Remaining: 8) at time 44
Process 7 completed at time 52, Waiting Time: 43, Turnaround Time: 51
Starting/Resuming Process 2 (Remaining: 9) at time 52
Process 2 completed at time 61, Waiting Time: 52, Turnaround Time: 61

--- SRTF Summary ---
Average Waiting Time: 17.36
Average Turnaround Time: 21.71

--- Round Robin Scheduling ---

--- Round Robin Scheduling (Time Quantum = 2) ---
Starting/Resuming Process 1 at time 0 (Remaining: 4)
Process 1 used its time quantum, remaining: 2
Starting/Resuming Process 2 at time 2 (Remaining: 9)
Process 2 used its time quantum, remaining: 7
Starting/Resuming Process 3 at time 4 (Remaining: 3)
Process 3 used its time quantum, remaining: 1
```

Tom Steinman
3/3/2025

```
Starting/Resuming Process 1 at time 6 (Remaining: 2)
Process 1 completed at time 8, Waiting Time: 4, Turnaround Time: 8
Starting/Resuming Process 7 at time 8 (Remaining: 8)
Process 7 used its time quantum, remaining: 6
Starting/Resuming Process 9 at time 10 (Remaining: 4)
Process 9 used its time quantum, remaining: 2
Starting/Resuming Process 8 at time 12 (Remaining: 5)
Process 8 used its time quantum, remaining: 3
New transient event (process 999) arrived at time 15!
Process Details: Duration=9, Priority=1
Starting Transient Process 999 at time 14
Process 999 completed at time 23, Waiting Time: 0, Turnaround Time: 9
Starting/Resuming Process 13 at time 23 (Remaining: 7)
Process 13 used its time quantum, remaining: 5
Starting/Resuming Process 2 at time 25 (Remaining: 7)
Process 2 used its time quantum, remaining: 5
Starting/Resuming Process 4 at time 27 (Remaining: 1)
Process 4 completed at time 28, Waiting Time: 23, Turnaround Time: 24
Starting/Resuming Process 5 at time 28 (Remaining: 4)
Process 5 used its time quantum, remaining: 2
Starting/Resuming Process 3 at time 30 (Remaining: 1)
Process 3 completed at time 31, Waiting Time: 28, Turnaround Time: 31
Starting/Resuming Process 6 at time 31 (Remaining: 2)
Process 6 completed at time 33, Waiting Time: 26, Turnaround Time: 28
Starting/Resuming Process 11 at time 33 (Remaining: 4)
Process 11 used its time quantum, remaining: 2
Starting/Resuming Process 12 at time 35 (Remaining: 4)
Process 12 used its time quantum, remaining: 2
Starting/Resuming Process 7 at time 37 (Remaining: 6)
Process 7 used its time quantum, remaining: 4
Starting/Resuming Process 10 at time 39 (Remaining: 2)
Process 10 completed at time 41, Waiting Time: 30, Turnaround Time: 32
Starting/Resuming Process 9 at time 41 (Remaining: 2)
Process 9 completed at time 43, Waiting Time: 38, Turnaround Time: 42
Starting/Resuming Process 8 at time 43 (Remaining: 3)
Process 8 used its time quantum, remaining: 1
Starting/Resuming Process 13 at time 45 (Remaining: 5)
Process 13 used its time quantum, remaining: 3
Starting/Resuming Process 2 at time 47 (Remaining: 5)
Process 2 used its time quantum, remaining: 3
Starting/Resuming Process 5 at time 49 (Remaining: 2)
Process 5 completed at time 51, Waiting Time: 43, Turnaround Time: 47
Starting/Resuming Process 11 at time 51 (Remaining: 2)
Process 11 completed at time 53, Waiting Time: 42, Turnaround Time: 46
```

```
Starting/Resuming Process 12 at time 53 (Remaining: 2)
Process 12 completed at time 55, Waiting Time: 43, Turnaround Time: 47
Starting/Resuming Process 7 at time 55 (Remaining: 4)
Process 7 used its time quantum, remaining: 2
Starting/Resuming Process 8 at time 57 (Remaining: 1)
Process 8 completed at time 58, Waiting Time: 51, Turnaround Time: 56
Starting/Resuming Process 13 at time 58 (Remaining: 3)
Process 13 used its time quantum, remaining: 1
Starting/Resuming Process 2 at time 60 (Remaining: 3)
Process 2 used its time quantum, remaining: 1
Starting/Resuming Process 7 at time 62 (Remaining: 2)
Process 7 completed at time 64, Waiting Time: 55, Turnaround Time: 63
Starting/Resuming Process 13 at time 64 (Remaining: 1)
Process 13 completed at time 65, Waiting Time: 56, Turnaround Time: 63
Starting/Resuming Process 2 at time 65 (Remaining: 1)
Process 2 completed at time 66, Waiting Time: 57, Turnaround Time: 66

--- Round Robin Summary ---
Average Waiting Time: 35.43
Average Turnaround Time: 40.14
```



```
--- Comparison of Scheduling Algorithms ---
Algorithm       Avg Waiting Time     Avg Turnaround Time
------------------------------------------------------------
FCFS            32.43                37.21
SJF             15.43                19.57
Priority        26.93                31.14
SRTF            17.36                21.71
Round Robin     35.43                40.14
```

Algorithm Comparisons

One thing that I didn’t take into account was making the transient event random. This makes it a bit more difficult to determine which algorithm would perform best, as each algorithm has a transient event of different attributes. The transient event attributes should be static for better evaluation of the algorithms. In this case however, the Shortest Job First algorithm had the best performance. This is likely due to the fact that my processes had a duration between 1-10, allowing SJF to have the best average waiting and turnaround time. The next best was SRTF, which was surprising as I expected it to perform close or better than SJF. Next was priority scheduling, as completing processes with a higher priority usually causes a longer waiting time. FCFS was fourth, and likely had poor performance due to shorter processes being stuck behind longer processes (convoy effect). In this case, the arrival times and process durations seemed to have a detrimental effect on the FCFS algorithm. Finally, the Round Robin algorithm was the worst. This is likely due to choosing a time quantum of 2 was probably not the best choice for the processes chosen. However, the main benefits of RR are its attributes of fairness and preventing starvation so the it being last was not incredibly surprising.

Pros and Cons of Scheduling Algorithms

First Come First Served (FCFS)

Pros:

- Simple to implement and understand, with no complex computations
- Fair in terms of arrival time (processes are executed in the exact order they arrive)
- No starvation (every process eventually gets CPU time)
- Good for workloads without strict time constraints

Cons:

- Poor average waiting time, especially when short processes are behind long ones (convoy effect)
- Not responsive to high-priority tasks or short tasks
- Not optimal for interactive systems where quick response time is needed
- Poor handling of transient events that need immediate attention (has to wait for current process to complete)

Shortest Job First (SJF)

Pros:

- Optimal average waiting time among non-preemptive algorithms
- Good for batch systems where process durations are known
- Prioritizes quick tasks, which can improve system responsiveness
- Better handling of short transient events than FCFS

Cons:

- Requires knowledge of process execution time in advance (often not available in real systems)
- Potential for starvation of longer processes if short processes keep arriving
- Long processes may experience significant delays
- Not ideal for interactive or real-time systems

Priority Scheduling

Pros:

- Allows important processes to be executed first
- Flexible prioritization based on various metrics (not just duration)
- Useful in real-time systems where certain tasks have strict deadlines
- Can handle critical transient events by assigning them high priority rather than hard coding it like in my implementation

Cons:

- Potential for starvation of low-priority processes
- Complexity in determining appropriate priority values
- Priority inversion problems (low-priority process holding resources needed by high-priority process)
- System overhead in maintaining priority queues

Shortest Remaining Time First (SRTF)

Pros:

- Optimal average waiting time among all scheduling algorithms
- Preemptive, so it can respond quickly to short processes
- Excellent for minimizing average waiting time
- Highly responsive to short transient events

Cons:

- Requires continuous monitoring of remaining process time
- High overhead due to frequent context switching
- Potential starvation of longer processes
- Difficult to implement in practice as execution times are typically unknown

Round Robin

Pros:

- Fair allocation of CPU time to all processes
- No starvation as each process gets a time slice
- Good for time-sharing systems
- Predictable response times (bounded by time quantum * number of processes)
- Handles transient events within a reasonable time frame

Cons:

- Performance heavily dependent on choice of time quantum
- If time quantum too small: excessive context switching overhead

- If time quantum too large: degenerates to FCFS
- Not optimal for systems with widely varying process times
- Average waiting time not as good as SJF or SRTF