Operating Systems Lab Project Report

➔ Crew of the Project:

- AP19110010221 - Tankala Yuvaraj.
- AP19110010192 - K. Subrahmanyam.
- AP19110010174 - Boyapati Sai Venkat.
- AP19110010150 - D. Nithya.
- AP19110010171 - A. Likhitha.

Title: Develop/Simulate a preemptive scheduling algorithm

SRM UNIVERSITY AP ANDHRA PRADESH

COMPUTER SCIENCE AND ENGINEERING

Declaration And Approval:

We, students from SRM-AP college, Amaravathi hereby declare that the project work entitled “Develop/Simulate a preemptive scheduling algorithm” has been submitted to our project organizer.

It is a record of original work done by our group, we also declare that this project Report is of our own effort and it has not been submitted to any other university for the award of any degree.

Student’s Signature,

Tankala Yuvaraj,

K. Subrahmanyam,

Boyapati Sai Venkat,

D. Nithya,

A. Likhitha,

Table of content

Subject: Operating System

Lab Project Work

Acknowledgment

We would like to acknowledge everyone involved in making the internship experience for us a smooth learning curve and helping in the completion of this Semester End Lab Project.

We would like to express our sincere gratitude to our professor's Ashu Abdul sir (Operating Systems), who guided us further in the successful completion of this project "Developing preemptive scheduling algorithm'' which also helped us in learning new concepts. Secondly, we would like to thank our group members for being so coordinated and helping a lot to finish this project.

⟶ Project Title:

➔ Develop/Simulate a preemptive scheduling algorithm

Abstract

The primary objectives of scheduling algorithms are to reduce resource depletion and ensure equality among those who use the resources. The difficulty of selecting which of the pending requests should be given resources is dealt with by scheduling. We have a manual computation process, but to make it easier, we developed a simulator using the Django web application and the existing preemptive and non-preemptive algorithms. The main purpose of this web application is to provide a platform that can help us to save our efforts and time.

Body of the project

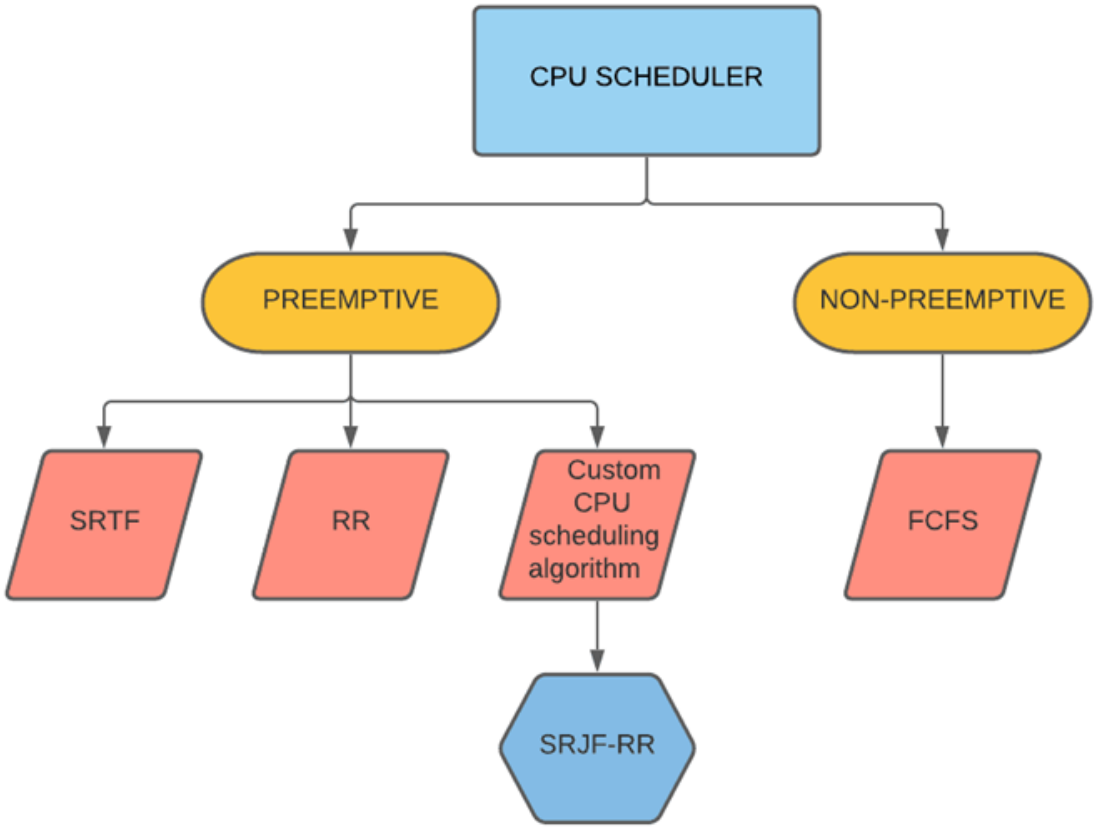
Our group came up with intending to design a project to create a web application using Django to implement the preemptive scheduling algorithms. As well as a new preemptive scheduling algorithm that combines both Round Robin(RR) and Shortest Remaining Time First(SRTF) algorithms, primarily to improve the efficiency of the existing RR algorithm. To further about the new preemptive algorithm, it is even more effective in terms of computation and average turnaround time than the conventional RR process. Waiting time and context switching are both factors in play.

The main objective and use of this web application is the user who accesses our website and implements the needful algorithms by electing according to their need

by selecting them. After the user can enter the values of arrival time, burst time, time quantum, etc., So that by the user data which they entered the implementation happens and the need outcomes line Gantt chart, output values table will be displayed respectively.

With the help of Django, web application users can use this as a compiler or as a calculator to know the outputs values for the given data of the scheduling algorithms.

Introduction to concepts used:



Schedulers:

- Wherever the CPU becomes idle, the system must select one of the processes which are ready in the queue to be executed. The selection process is carried out by the CPU Scheduler. The schedular selects a process from the processes in memory and that are ready to execute and allocates the CPU to that process.
- CPU scheduling decisions may take place when a process:
    1. Switches from running to waiting for state
    2. Switches from running to ready state
    3. Switches from waiting to ready
    4. Terminates
- Scheduling techniques under “1” and “4” are Non-Preemptive
- All other scheduling techniques are Preemptive
    - Consider access to shared data
    - Consider preemption while in kernel mode
    - Consider interrupts occurring during crucial OS activities.
- As we discussed above the scheduling algorithms are two types they are:
    - Non-Preemptive scheduling.
    - Preemptive scheduling.

Non-Preemptive scheduling:

Non-preemptive When a process terminates or transitions from the running to the waiting state, scheduling is used. Once the resources (CPU cycles) are allocated to a process, the process retains control of the CPU until it is terminated or reaches a waiting state.

Non-preemptive scheduling does not interrupt a running CPU process in the middle of its execution. Instead, it waits until the process's CPU burst time is over before allocating the CPU to another process.

Preemptive scheduling:

When a process transitions from the running state to the ready state or from the waiting state to the ready state, preemptive scheduling is used. The resources (primarily CPU cycles) are allocated to the process for a limited time before being removed, and the process is placed back in the ready queue if it still has CPU burst time remaining. That process remains in the ready queue until it is given another opportunity to execute.

The main key differences between the Preemptive and Non-Preemptive Scheduling Algorithms:

Parameter	Preemptive Scheduling	Non-preemptive Scheduling
Basic	In this resources(CPU Cycle) are allocated to a process for a limited time.	Once resources(CPU Cycle) are allocated to a process, the process holds it till it completes its burst time or switches to waiting for the state.
Interrupt	The process can be interrupted in between.	The process can not be interrupted until it terminates itself or its time is up.
Starvation	If a process has high priority frequently arrives in the ready queue, low priority The process may starve.	If a process with a long burst time is running CPU, then later coming process with less CPU burst time may starve.
Overhead	It has overheads of scheduling the processes.	It does not have overheads.
Flexibility	flexible	rigid
Cost	cost associated	no cost associated
CPU Utilization	In preemptive scheduling, CPU utilization is high.	It is low in non-preemptive scheduling.
Examples	Round Robin and Shortest Remaining Time First.	First Come First Serve and Shortest Job First.

First Come First Serve (FCFS):

- Simplest scheduling algorithm that schedules according to arrival times of processes.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- The first come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first.
- It is implemented by using the FIFO queue.
- When a process enters the ready queue, its PCB is linked to the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.
- FCFS is a non-preemptive scheduling algorithm.

Shortest Remaining Time First (SRTF):

- It is a preemptive mode of the SJF algorithm in which jobs are scheduled according to the shortest remaining time.
- The SJF algorithm can be either preemptive or non-preemptive.
- The choice arises when a new process arrives at the ready queue while a previous process is still executing.
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.
- A preemptive SJF algorithm will preempt the currently executing process.

- A non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.
- Preemptive SJF scheduling is also called shortest-remaining-time-first scheduling.

Round Robin Scheduling:

- Each process is assigned a fixed time(Time Quantum/Time Slice) in a cyclic way.
- It is designed especially for the time-sharing system.
- The ready queue is treated as a circular queue.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1-time quantum.
- To implement Round Robin scheduling, we keep the ready queue as a FIFO queue of processes.
- New processes are added to the tail of the ready queue.
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after a 1-time quantum, and dispatches the process.
- One of two things will then happen.
    - The process may have a CPU burst of less than 1-time quantum.
    - In this case, the process itself will release the CPU voluntarily.
    - The scheduler will then proceed to the next process in the ready queue.

- Otherwise, if the CPU burst of the currently running process is longer than 1-time quantum, the timer will go off and will cause an interrupt to the operating system.
- A context switch will be executed, and the process will be put at the tail of the ready queue.
- The CPU scheduler will then select the next process in the ready queue.

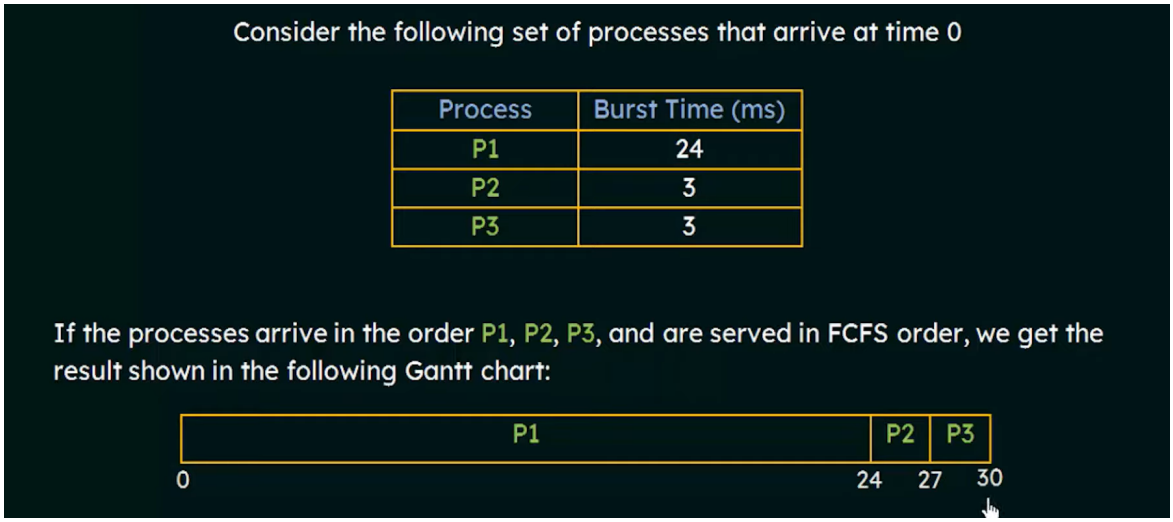
Different Time With Respect To A Process

- **Burst Time:** The burst time is the amount of time required by a process for executing on a CPU
- **Arrival Time:** Time at which the process arrives in the ready queue.
- **Completion Time:** Time at which the process completes its execution.
- **Turn Around Time:** Time Difference between completion time and arrival time.
  - Turn Around Time = Completion Time – Arrival Time
  - Turn Around time = Burst time + Waiting time
- **Waiting Time(W.T):** Time Difference between turn around time and burst time.
  - Waiting Time = Turn Around Time – Burst Time
  - Waiting Time = Service Time - Arrival Time
- **Response Time:** Response time is the amount of time after which a process gets the CPU for the first time after entering the ready queue.
  - Response Time = Time at which process first gets the CPU – Arrival time.

Gantt Chart

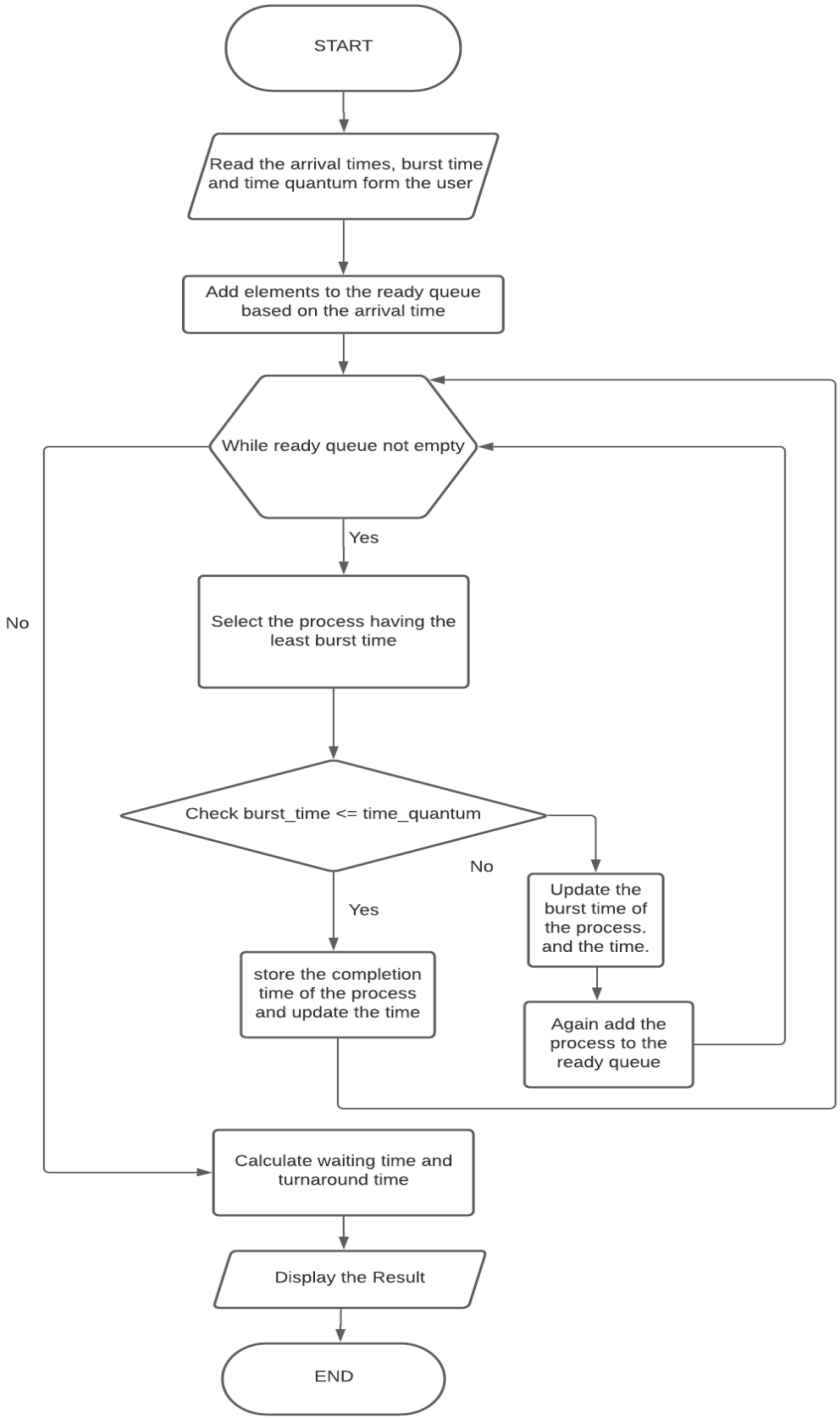
Generalized Activity Normalization Time Table (GANTT) chart is a type of chart in which a series of horizontal lines are present that show the amount of work done or production completed in a given period of time in relation to the amount planned for those projects.

Example:



Flow Chart

→ SRJF-RR Scheduling Algorithm



Code

```
# Create your views here.
def home(request):
    return render(request,'Main-page.html')
def index(request):
    d= {'FCFS': 'FCFS (First Come First Serve)','SRTF':'SRTF,Shortest
Remaining Time First (Preemtive)',"RR":"Round Robin
(Preemtive)","SRJFRR":"SRJFRR"}
    c={'d':d}
    return render(request,'home.html',c)
def add(request):
    a=(request.POST['a']).split()
    b=(request.POST['b']).split()
    d=int((request.POST['d']))
    e=(request.POST['lan'])
    for i in range(len(a)):
        a[i] = int(a[i])
        b[i] = int(b[i])
    arrival = a
    burst_time = b
    quantum_time = d
    n = len(a)
    gantt = []
    if e == 'FCFS': # FCFS Scheduling
        d_c = {} # Completion time
        d_b = {} # Burst time
```

```
        d_a = {} # Arrival time
        process_sno = [i for i in range(1,n+1)] # Storing each process
number
        t_a_t = [-1]*n  # Intializing a list of n element
        waiting_time = [-1]*n # Intializing a list of n elements
        completed = []
        gantt = []
        # Sorting the arrival time using bubble sort.
        for i in range(n-1):
                for j in range(0, n-i-1):
                    if arrival[j] > arrival[j + 1] :
                        arrival[j], arrival[j + 1] = arrival[j + 1],
arrival[j]
                        process_sno[j],process_sno[j+1] =
process_sno[j+1],process_sno[j]
                        burst_time[j],burst_time[j+1] =
burst_time[j+1],burst_time[j]
        for i in range(n):
            d_a[i+1] = arrival[i] # Key = process number , Value = arrival
time of that particular process
            d_b[i+1] = burst_time[i] # Key = process number , Value =
arrival time of that particular process
            d_c[i+1] = 0 # Key = process number , Value = arrival time of
that particular process
        ready_queue = [] # A ready_queue list  is used to store that burst
times of the processes.
        visited = [] # The element is add to the visited list.
        count = 0# Time
        for i in range(n):
```

```
            if i+1 not in visited and count >= d_a[i+1]: # A condition
checking whether the current time is greater than the arrival time of the
process.
                visited.append(i+1)  # The element is add to the visited
list.
                ready_queue.append(i+1) # The corresponding process number
is add to this list
        while ready_queue: # Condition if ready queue is empty or not.
            m = ready_queue.pop(0) # The process is taken on the FCFS
basis.
            process_number = m
            b_time = d_b[m] # Burst time of process
            d_c[process_number] = count + d_b[m]  # Completion time
calculation
            count = d_c[process_number] # Update the current time
            for i in range(n):
                if i+1 not in visited and count >= d_a[i+1]: # A condition
checking whether the current time is greater than the arrival time of the
process.
                    visited.append(i+1)  # The element is add to the
visited list.
                    ready_queue.append(i+1) # The corresponding process
number is add to this list
        gantt = process_sno # gantt chart
        # Turn Around Time.
        for i in range(n):
            t_a_t[i] = d_c[i+1] - d_a[i+1]
        # Burst Time.
        for i in range(n):
```

```
            waiting_time[i] = t_a_t[i] - burst_time[i]
        Average_waiting_time = sum(waiting_time) / n
        Average_turnaround_time = sum(t_a_t) / n
    elif e == "SRTF":
        d_c = {} # Completion time
        d_b = {} # Burst time
        d_a = {} # Arrival time
        process_sno = [i for i in range(1,n+1)] # Storing each process
number
        t_a_t = [-1]*n  # Intializing a list of n element
        waiting_time = [-1]*n # Intializing a list of n elements
        completed = [] # Initializing a list that is used to tell which
processes are completed
        for i in range(n): # A loop for storing arrival , burst times and
process numbers in the form of key value pairs.
            d_a[i+1] = arrival[i] # Key = process number , Value = arrival
time of that particular process
            d_b[i+1] = burst_time[i] # Key = process number , Value =
arrival time of that particular process
            d_c[i+1] = 0 # Key = process number , Value = arrival time of
that particular process
        ready_queue = [] # A ready_queue list  is used to store that burst
times of the processes.
        rq = [] # A similar queue to ready queue to store the particular
process number for all the queue
        visited = [] # A list of all the visited processes.
        count = 0# Time
        for i in range(n):
```

```
            if i+1 not in visited and count >= d_a[i+1]:  # A condition
checking whether the current time is greater than the arrival time of the
process.
                visited.append(i+1) # The element is add to the visited
list.
                ready_queue.append(d_b[i+1]) # The burst time of the
process in appended to the ready queue.
                rq.append(i+1) # The corresponding process number is add
to this list
        while ready_queue: # Condition if ready queue is empty or not.
            m = min(ready_queue) # The process having the minimum burst
time is found.
            idx = ready_queue.index(m) # Index of the minimum burst time
process is computed.
            process_number = rq[idx] # Corresponding process number is
stored in this variable using the index that is found.
            ready_queue.remove(m) # The process's burst time is removed
from the ready queue.
            rq.pop(idx) # The corresponding process number is removed from
this list.
            if m <= 1: # m = burst time of process , if the burst time is
less than or equal to 1 , then the process is said to be completed.
                d_c[process_number] = count + 1 # Count = Time , The time
when the process is completed is stored in the d_c dictionary.
                completed.append(process_number) # Process number is added
to the completed list
                count = d_c[process_number] # Update the time
            else:
                count += 1  # Update the time
                d_b[process_number] -= 1 # Update the burst time of the
process
```

```
            for i in range(n):
                if i+1 not in visited and count >= d_a[i+1]:  # A
condition checking whether the current time is greater than the arrival
time of the process.
                    visited.append(i+1) # The element is add to the
visited list.
                    ready_queue.append(d_b[i+1]) # The burst time of the
process in appended to the ready queue.
                    rq.append(i+1) # The corresponding process number is
add to this list
            if process_number not in completed:  # If the process is not
completed, it is again added to the ready queue.
                ready_queue.append(d_b[process_number]) # Adding the
updated burst time of the particular process to the ready queue.
                rq.append(process_number) # Adding the process number.
            if not gantt or gantt[-1] != process_number: # if the previous
process is also the same process that is being executed now, then there is
not need of duplicates.
                gantt.append(process_number) # Adding the process to the
gantt chart for displaying purposes.
        # Turn Around Time Computation TAT = CT - AT
        for i in range(n):
            t_a_t[i] = d_c[i+1] - d_a[i+1]
        # Waiting Time Computation.  WT = TAT - BT
        for i in range(n):
            waiting_time[i] = t_a_t[i] - burst_time[i]
        Average_waiting_time = sum(waiting_time) / n
        Average_turnaround_time = sum(t_a_t) / n
        pass
    elif e == "RR":
```

```
        d_c = {} # Completion time
        d_b = {} # Burst time
        d_a = {} # Arrival time
        process_sno = [i for i in range(1,n+1)] # Storing each process
number
        t_a_t = [-1]*n  # Intializing a list of n element
        waiting_time = [-1]*n # Intializing a list of n elements
        completed = [] # Initializing a list that is used to tell which
processes are completed
        for i in range(n): # A loop for storing arrival , burst times and
process numbers in the form of key value pairs.
            d_a[i+1] = arrival[i] # Key = process number , Value = arrival
time of that particular process
            d_b[i+1] = burst_time[i] # Key = process number , Value =
arrival time of that particular process
            d_c[i+1] = 0 # Key = process number , Value = arrival time of
that particular process
        ready_queue = [] # A ready_queue list  is used to store that burst
times of the processes.
        visited = []  # A list of all the visited processes.
        count = 0# Time
        for i in range(n):
            if i+1 not in visited and count >= d_a[i+1]: # A condition
checking whether the current time is greater than the arrival time of the
process.
                visited.append(i+1) # The element is add to the visited
list.
                ready_queue.append(i+1) # The corresponding process number
is add to this list
        while ready_queue: # Condition if ready queue is empty or not.
```

```
            m = ready_queue.pop(0) # The process is taken on the FCFS
basis.
            process_number = m # Process number.
            b_time = d_b[process_number] # Burst time of the process based
on the process number.
            if b_time <= quantum_time: # Checks if the burst time is less
than or equal to the quantum time.
                d_c[process_number] = count + b_time # Count = Time , The
time when the process is completed is stored in the d_c dictionary.
                completed.append(process_number) # Process number is added
to the completed list
                count = d_c[process_number] # Update the time
            else:
                count += quantum_time # Update the time
                d_b[m] -= quantum_time # Update the burst time of the
process
            for i in range(n):
                if i+1 not in visited and count >= d_a[i+1]: # A condition
checking whether the current time is greater than the arrival time of the
process.
                    visited.append(i+1)  # The element is add to the
visited list.
                    ready_queue.append(i+1) # The corresponding process
number is add to this list
            if process_number not in completed:  # If the process is not
completed, it is again added to the ready queue.
                ready_queue.append(process_number) # Adding the updated
burst time of the particular process to the ready queue.
            if not gantt or gantt[-1] != process_number: # if the previous
process is also the same process that is being executed now, then there is
not need of duplicates.
```

```
                gantt.append(process_number) # Adding the process to the
gantt chart for displaying purposes.
        # Turn Around Time Computation
        for i in range(n):
            t_a_t[i] = d_c[i+1] - d_a[i+1]
        # Waiting Time Computation.  WT = TAT - BT
        for i in range(n):
            waiting_time[i] = t_a_t[i] - burst_time[i]
        Average_waiting_time = sum(waiting_time) / n
        Average_turnaround_time = sum(t_a_t) / n
        pass
    elif e == "SRJFRR":
        d_c = {} # Completion time
        d_b = {} # Burst time
        d_a = {} # Arrival time
        process_sno = [i for i in range(1,n+1)] # Storing each process
number
        t_a_t = [-1]*n  # Intializing a list of n element
        waiting_time = [-1]*n # Intializing a list of n elements
        completed = [] # Initializing a list that is used to tell which
processes are completed
        for i in range(n):  # A loop for storing arrival , burst times and
process numbers in the form of key value pairs.
            d_a[i+1] = arrival[i] # Key = process number , Value = arrival
time of that particular process
            d_b[i+1] = burst_time[i] # Key = process number , Value =
arrival time of that particular process
            d_c[i+1] = 0 # Key = process number , Value = arrival time of
that particular process
```

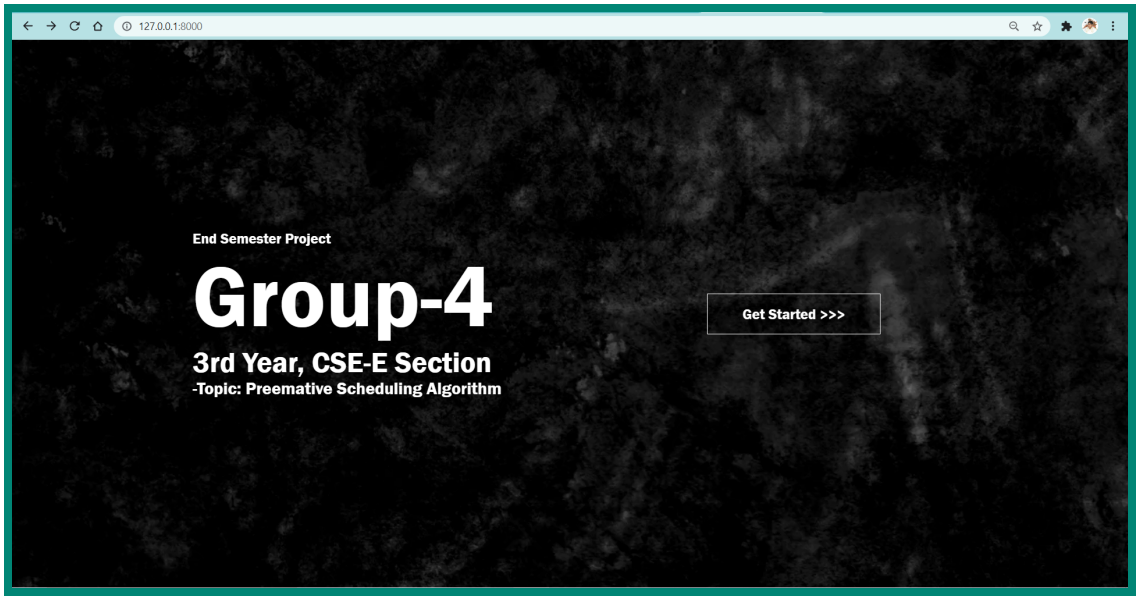
```
        ready_queue = []  # A ready_queue list  is used to store that
burst times of the processes.
        rq = [] # A similar queue to ready queue to store the particular
process number for all the queue
        visited = []  # A list of all the visited processes.
        count = 0# Time
        for i in range(n):
            if i+1 not in visited and count >= d_a[i+1]: # A condition
checking whether the current time is greater than the arrival time of the
process.
                visited.append(i+1) # The element is add to the visited
list.
                ready_queue.append(d_b[i+1]) # The corresponding process
number is add to this list
                rq.append(i+1)  # The corresponding process number is add
to this list
        while ready_queue:  # Condition if ready queue is empty or not.
            m = min(ready_queue) # The process having the minimum burst
time is found.
            idx = ready_queue.index(m) # Index of the minimum burst time
process is computed.
            process_number = rq[idx] # Corresponding process number is
stored in this variable using the index that is found.
            ready_queue.remove(m) # The process's burst time is removed
from the ready queue.
            rq.pop(idx) # The corresponding process number is removed from
this list.
            if m <= quantum_time: # m = burst time of process , if the
burst time is less than or equal to 1 , then the process is said to be
completed.
```

```
                d_c[process_number] = count + d_b[process_number] # Count
= Time , The time when the process is completed is stored in the d_c
dictionary.
                completed.append(process_number) # Process number is added
to the completed list
                count = d_c[process_number]  # Update the time
            else:
                count += quantum_time # Update the time
                d_b[process_number] -= quantum_time  # Update the burst
time of the process
            for i in range(n):
                if i+1 not in visited and count >= d_a[i+1]:
                    visited.append(i+1) # The element is add to the
visited list.
                    ready_queue.append(d_b[i+1]) # The burst time of the
process in appended to the ready queue
                    rq.append(i+1) # The corresponding process number is
add to this list
            if process_number not in completed:  # If the process is not
completed, it is again added to the ready queue.
                ready_queue.append(d_b[process_number]) # Adding the
updated burst time of the particular process to the ready queue.
                rq.append(process_number) # Adding the process number.
            if not gantt or gantt[-1] != process_number: # if the previous
process is also the same process that is being executed now, then there is
not need of duplicates.
                gantt.append(process_number) # Adding the process to the
gantt chart for displaying purposes.
        # Turn Around Time Computation
        for i in range(n):
            t_a_t[i] = d_c[i+1] - d_a[i+1]
```

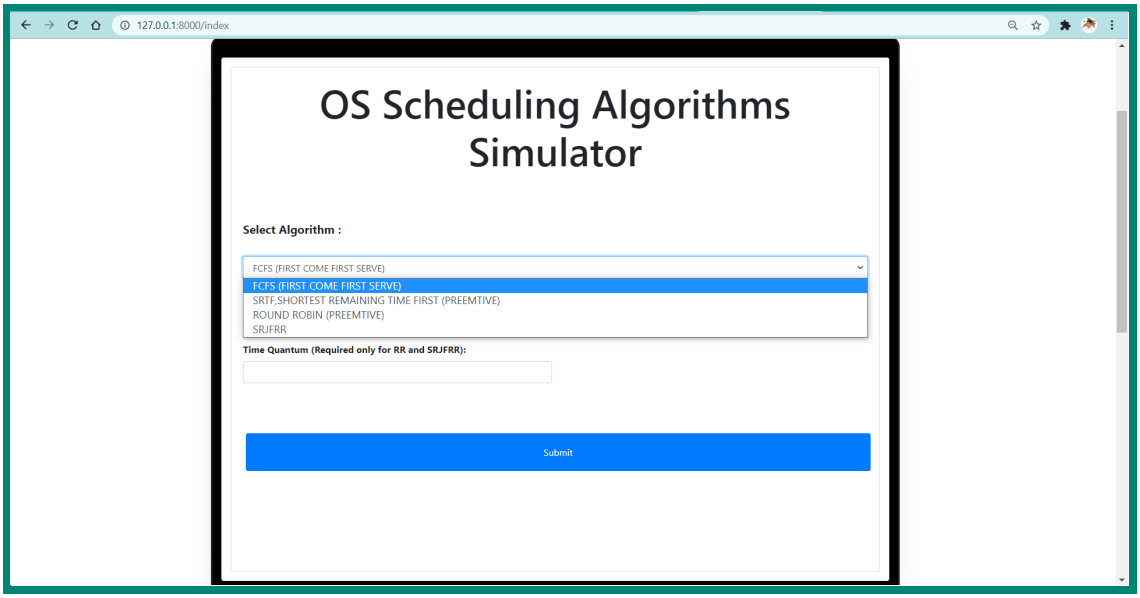
```
        # Waiting Time Computation.  WT = TAT - BT
        for i in range(n):
            waiting_time[i] = t_a_t[i] - burst_time[i]
        Average_waiting_time = sum(waiting_time) / n
        Average_turnaround_time = sum(t_a_t) / n
        pass
    data = []
    for i in range(n):
data.append([process_sno[i],arrival[i],burst_time[i],d_c[i+1],t_a_t[i],wai
ting_time[i]])
    print(data)
    print(gantt)
    res = {
        "data" : data,
        "Avg_t":Average_turnaround_time,
        "Avg_w":Average_waiting_time,
        "gantt":gantt
    }
    return render(request,'home.html',res)
```

Outputs

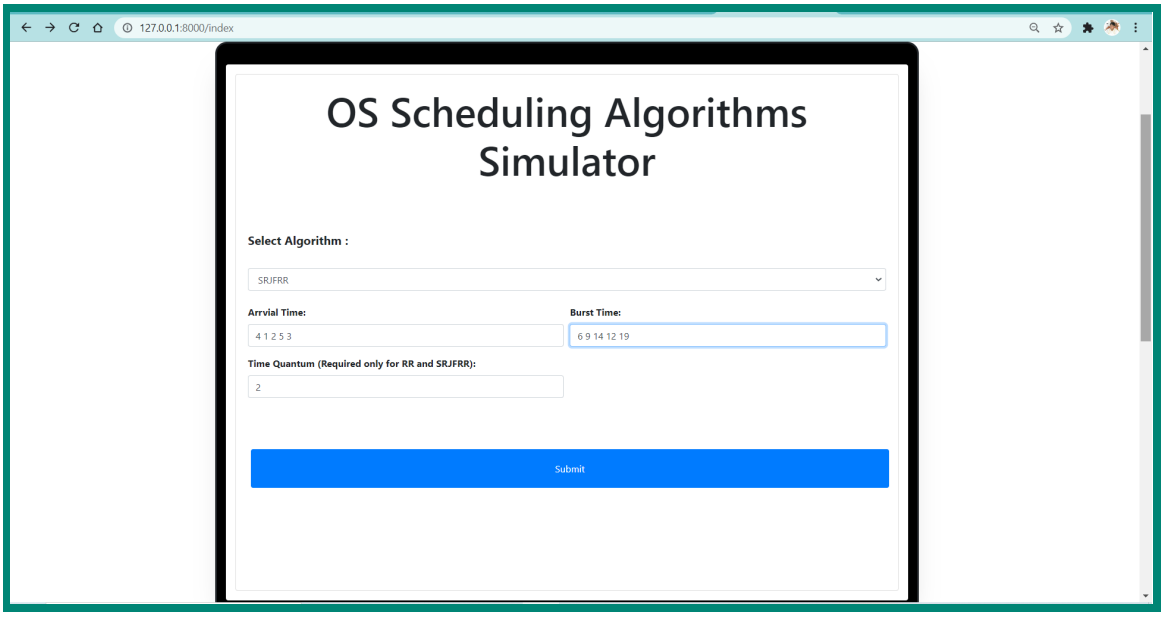
Main page:



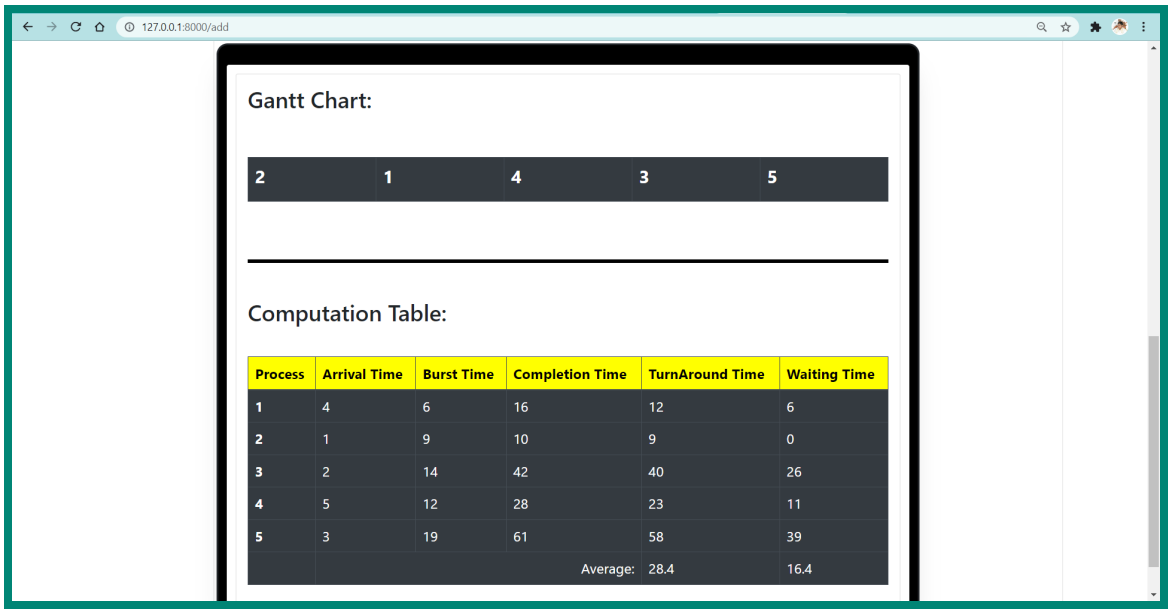
Select Option Page:



Example_Output:



Example Output Values:



Conclusion

An operating system uses process scheduling to make sure that every process which is ready to perform and allocates CPU to one of them. Objects of the Scheduling algorithm are: CPU resources wisely, increase throughput, reduce wait time, increase response and turnaround times. Compared to the Round Robin algorithm, the custom preemptive scheduling algorithm (SRJF-RR) has significantly reduced waiting time, turnaround time, and context switching.

GitHub Link

https://github.com/yuvaraj-06/OS-Project-4

References

- Textbook of Operating Systems: Operating System Concepts – 9th Edition Silberschatz, Galvin and Gagne ©2013
- https://www.geeksforgeeks.org/operating-systems/
- https://www.youtube.com/c/nesoacademy
- https://www.geeksforgeeks.org/preemptive-and-non-preemptive-scheduling/
- https://www.tutorialspoint.com/preemptive-and-non-preemptive-scheduling
- https://www.geeksforgeeks.org/cpu-scheduling-in-operating-systems-using-priority-queue-with-gantt-chart/
- https://www.javatpoint.com/os-cpu-scheduling