Dr. AMBEDKAR INSTITUTE OF TECHNOLOGY

An Autonomous Institute, Affiliated to Visvesvaraya Technological University, Belagavi. Accredited by N.A.A.C, with 'A' Grade)

Near Jnana Bharathi Campus, Malathahalli, Bengaluru-560 056, Karnataka, India

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Group Activity Report on

"Python implementation of Shortest Job First (SJF) CPU Scheduling"

Submitted in partial fulfillment of

Operating System
Course Code: 21CST404

Submitted by

Student Name	USN
Amruthamsh A	1DA21CS017
Angelina Miriam Denny	1DA21CS019
Anika Shetty	1DA21CS020
Chaitra N	1DA21CS037
Chandana S J	1DA21CS039

For the Academic Year 2022-23

Dr. Ambedkar Institute of Technology

((Near Jnana Bharathi Campus, Mallathahalli, Bengaluru – 560056)

Department of Computer Science & Engineering

CERTIFICATE

This is to certify that the report entitled has presented seminar entitled **"Python implementation of Shortest Job First (SJF) CPU Scheduling"** and submitted in partial fulfillment of the 4th semester activity curricular during the year 2022-23 is a result of genuine work carried out by the team of five students of 4th Semester CSE.

Signature of the professor　　　　　　Signature of the H.O.D

TABLE OF CONTENTS

1. Introduction 4

2. Methodology 5

2.1 Characteristics of SJF Scheduling

2.2 Examples to show working of SJF

2.3 Algorithm for Non-Preemptive SJF

2.4 Algorithm for Preemptive SJF

2.5 Environment for implementing SJF

3. Results 19

4. Conclusion 23

5. References 24

6. Source Code 25

1. Introduction

The efficient allocation of resources in computer systems is a critical concern for optimizing performance and ensuring effective task scheduling. One prominent algorithm that addresses this concern is the Shortest Job First (SJF) algorithm. SJF is a non-preemptive scheduling algorithm that selects the next task for execution based on the shortest burst time, or execution time. This algorithm aims to minimize the average waiting time and turnaround time for processes, resulting in improved system efficiency and responsiveness.

The core principle of the SJF algorithm revolves around executing the shortest job next, thereby reducing the time taken for each task to complete. In scenarios where tasks have varying burst times, SJF can help prevent longer processes from monopolizing system resources, leading to enhanced fairness and overall performance.

The implementation of scheduling algorithms like SJF often involves intricate calculations and comparisons. Leveraging the capabilities of programming languages and platforms can significantly simplify the development process. Python, renowned for its versatility and ease of use, is an ideal choice for creating such algorithms.

Additionally, for colaborative and interactive programming, Google Colab provides an excellent environment. Google Colab offers a cloud-based Jupyter Notebook interface that supports code execution, visualization, and data manipulation. The combination of Python and Google Colab empowers developers to seamlessly experiment with algorithmic implementations, visualize results, and share insights.

In this report, we will explore the Shortest Job First algorithm in detail, discussing its underlying principles and advantages. Furthermore, we will demonstrate how Python, with its expressive syntax, and Google Colab, with its colaborative capabilities, can be harnessed to implement and analyze the SJF algorithm effectively. By amalgamating theoretical concepts with practical implementation, we aim to comprehensively understand the SJF algorithm and the tools used to bring it to life.

In the subsequent sections, we will delve into the step-by-step implementation of the SJF algorithm using Python within the Google Colab environment. Through code examples, visual representations, and explanations, we intend to elucidate the algorithm's mechanics and showcase the benefits of utilizing modern programming tools.

2. Methodology

An approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the shortest-next-CPU-burst algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length. We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.

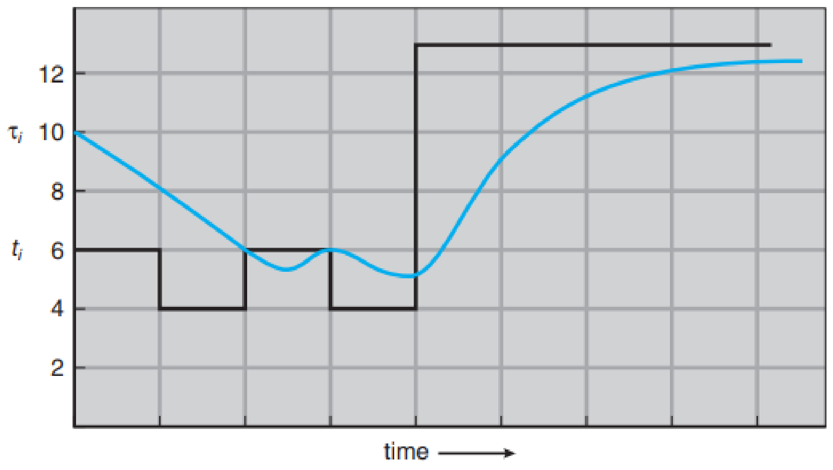
The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before long one decrease the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

Although the SJF algorithm is optimal, it cannot be implemented at the level of CPU scheduling, as there is no way to know the length of the next CPU burst. One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. We can define the exponential average with the following formula. Let tn be the length of the nth CPU burst, and let τn+1 be our predicted value for the next CPU burst. Then, for α, $0 \leq \alpha \leq 1$, define

$$\tau n+1 = \alpha \tau n + (1 - \alpha)\tau n$$

The value of tn contains our most recent information, while τn stores the past history. The parameter α controls the relative weight of recent and past history in our prediction. If $\alpha = 0$, then τn+1 = τn, and recent history has no effect (current conditions are assumed to be transient). If $\alpha = 1$, then τn+1 = tn, and only the most recent CPU burst matters (history is assumed to be old and irrelevant). More commonly, $\alpha = 1/2$, so recent history and past history are equally weighted. The initial τ0 can be defined as a constant or as an overall system average. Figure. 1 shows an exponential average with $\alpha = 1/2$ and $\tau 0 = 10$.



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Fig 1: Prediction of the length of the next CPU burst.

To understand the behavior of the exponential average, we can expand the formula for τ_{n+1} by substituting for τ_n to find

$$\tau_{n+1} = \alpha\tau_n + (1-\alpha)\alpha\tau_{n-1} + \cdots + (1-\alpha)^j \alpha\tau_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0$$

Typically, α is less than 1. As a result, $(1 - \alpha)$ is also less than 1, and each successive term has less weight than its predecessor.

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, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called shortest-remaining time-firs scheduling.

2.1 Characteristics of SJF Scheduling

Following are some characteristics of SJF:

- Associated with every job as it requires a unit of time for a job to complete.
- Helpful for batch-type processing.
- Improves process throughput as shorter jobs execute first, further reducing the turnaround time.
- Offers shorter jobs that improve job output.

2.1.1 Advantages of SJF

Following are the advantages of SJF:

- Used for long-term scheduling.
- Reduces average waiting time.
- Helpful for batch-type processing where runtimes are known in advance.
- For long-term scheduling, we can obtain a burst time estimate from the job description.
- It is necessary to predict the value of the next burst time for Short-Term Scheduling.
- Optimal with regard to average turnaround time.

2.1.2 Disadvantages of SJF

Following are the disadvantages of SJF:

- It is necessary to know the job completion time beforehand as it is hard to predict.
- Used for long-term scheduling in a batch system.
- Can’t implement this algorithm for CPU scheduling for the short term as we can’t predict the length of the upcoming CPU burst.
- Cause extremely long turnaround times or starvation.
- Knowledge about the runtime length of a process is necessary.
- The length of the upcoming CPU request is hard to predict.
- Recording the elapsed time results in increased overhead on the process

In case of Shortest Job First scheduling algorithm, the process with the smallest execution time gets executed next. There are two types of SJF: preemptive and non-preemptive. If offered shorter jobs, this algorithm can improve job output. It is a greedy algorithm that can cause starvation if only shorter jobs keep executing.

2.2 Examples to show working of SJF

The SJF algorithm aims to minimize the waiting time and turnaround time by scheduling processes with the shortest burst time first. Here we consider five processes each having its own unique burst time and arrival time to understand the working of SJF.

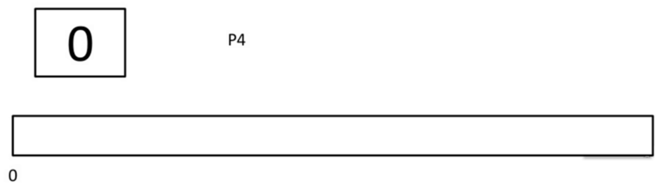
2.2.1 Non-Preemptive SJF

In non-preemptive scheduling, once the CPU cycle is allocated to process, the process holds it till it reaches a waiting state or terminated.

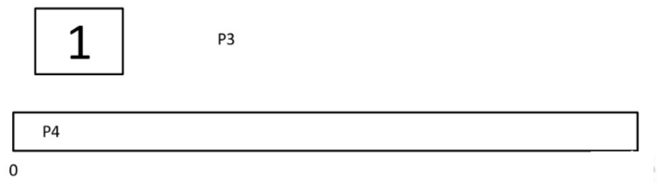
Consider the following five processes each having its own unique burst time and arrival time.

Process Queue	Burst time	Arrival time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

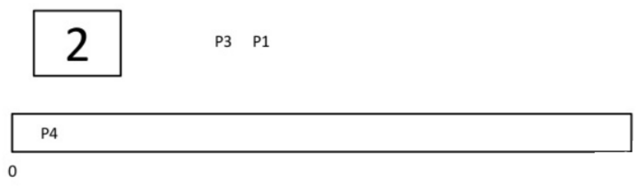
Step 0) At time=0, P4 arrives and starts execution.



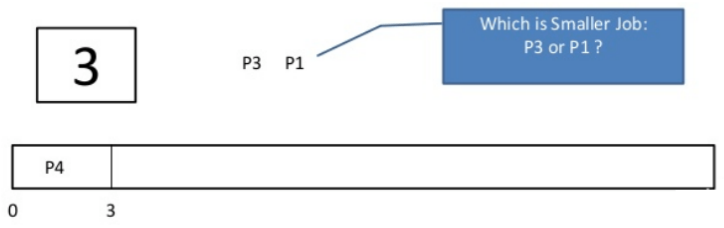
Step 1) At time= 1, Process P3 arrives. But, P4 still needs 2 execution units to complete. It will continue execution.



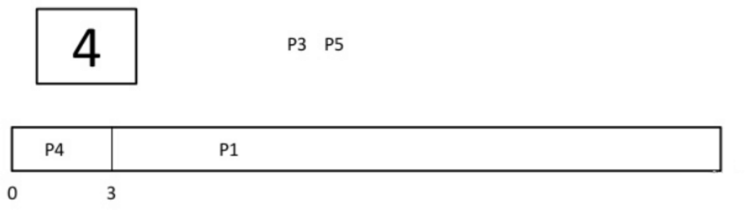
Step 2) At time =2, process P1 arrives and is added to the waiting queue. P4 will continue execution.



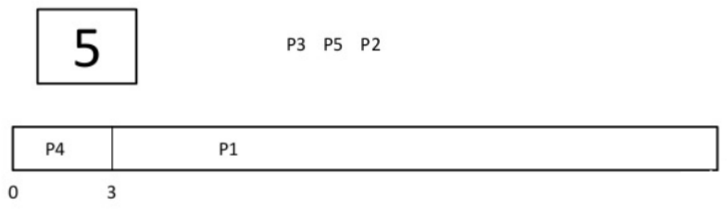
Step 3) At time = 3, process P4 will finish its execution. The burst time of P3 and P1 is compared. Process P1 is executed because its burst time is less compared to P3.



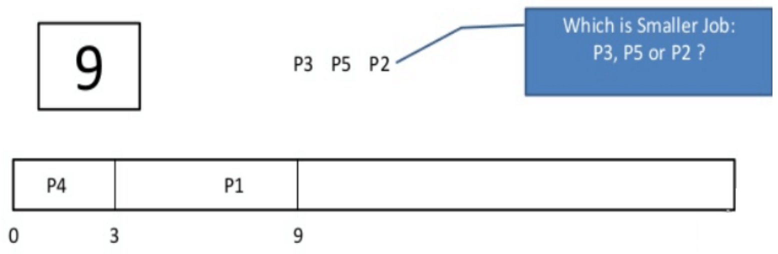
Step 4) At time = 4, process P5 arrives and is added to the waiting queue. P1 will continue execution.



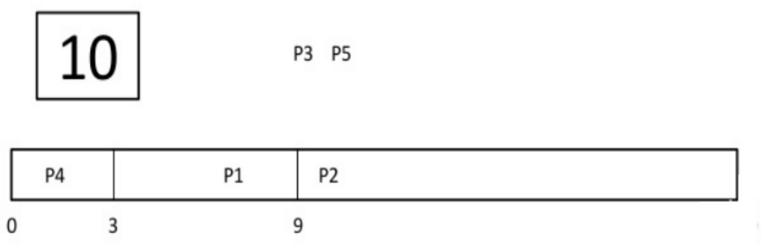
Step 5) At time = 5, process P2 arrives and is added to the waiting queue. P1 will continue execution.



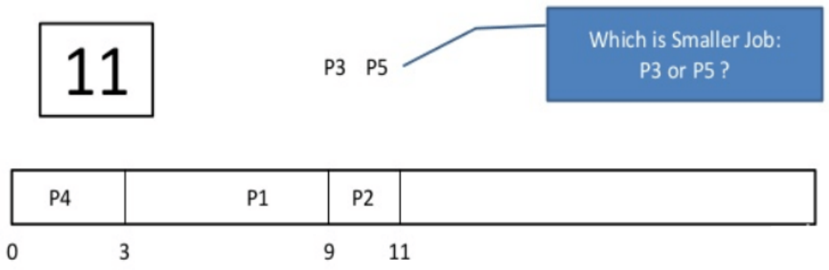
Step 6) At time = 9, process P1 will finish its execution. The burst time of P3, P5, and P2 is compared. Process P2 is executed because its burst time is the lowest.



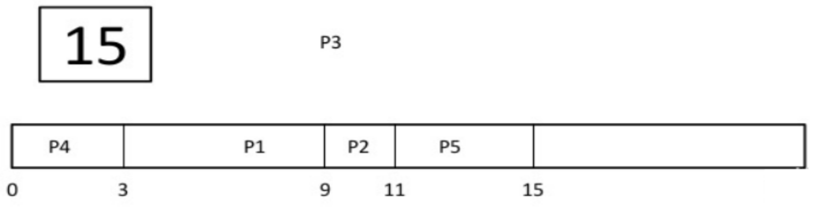
Step 7) At time=10, P2 is executing and P3 and P5 are in the waiting queue.



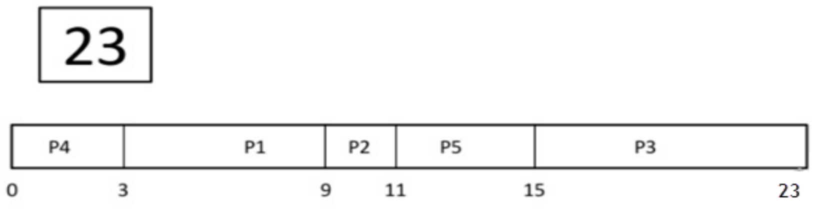
Step 8) At time = 11, process P2 will finish its execution. The burst time of P3 and P5 is compared. Process P5 is executed because its burst time is lower.



Step 9) At time = 15, process P5 will finish its execution.



Step 10) At time = 23, process P3 will finish its execution.



Step 11) Let’s calculate the average waiting time for above example.

Waiting time
P4= 0-0=0
P1= 3-2=1
P2= 9-5=4
P5= 11-4=7
P3= 15-1=14

Average Waiting Time= 0+1+4+7+14/5 = 26/5 = 5.2

2.2.2 Preemptive SJF

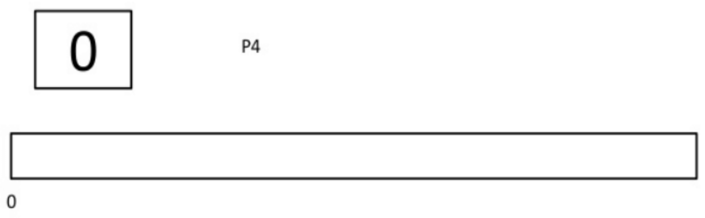
In Preemptive SJF Scheduling, jobs are put into the ready queue as they come. A process with shortest burst time begins execution. If a process with even a shorter burst time arrives, the current process is removed or preempted from execution, and the shorter job is allocated CPU cycle.

Consider the following five process:

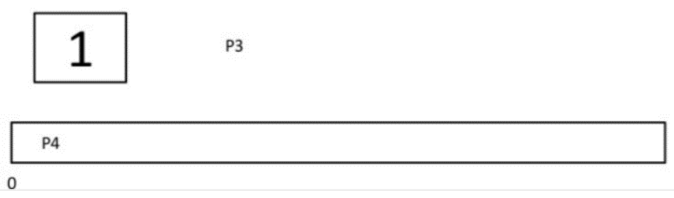
Process Queue	Burst time	Arrival time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

Step 0) At time=0, P4 arrives and starts execution.

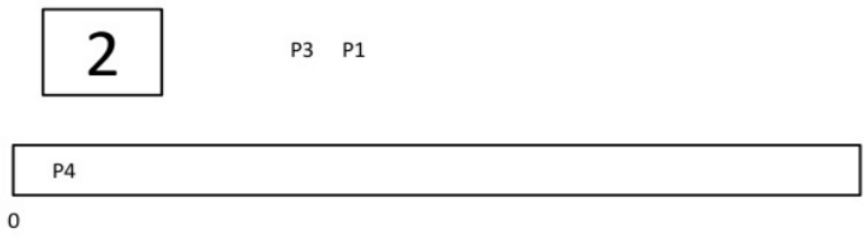
Process Queue	Burst time	Arrival time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4



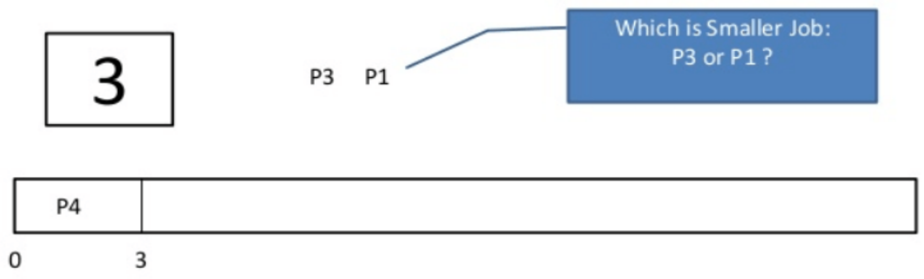
Step 1) At time= 1, Process P3 arrives. But, P4 has a shorter burst time. It will continue execution.



Step 2) At time = 2, process P1 arrives with burst time = 6. The burst time is more than that of P4. Hence, P4 will continue execution.

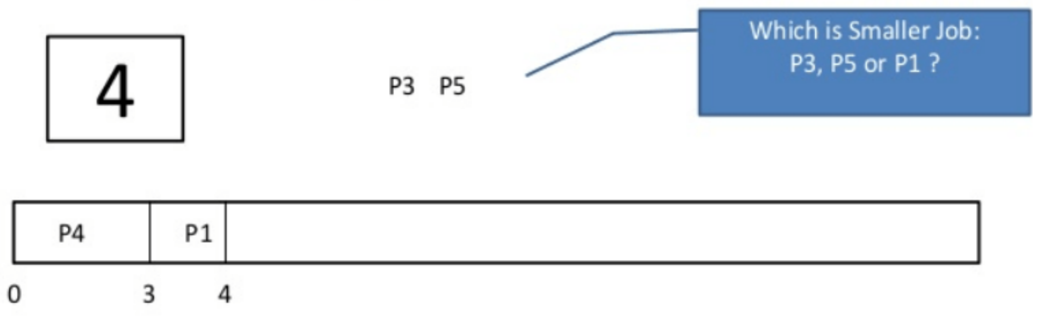


Step 3) At time = 3, process P4 will finish its execution. The burst time of P3 and P1 is compared. Process P1 is executed because its burst time is lower.



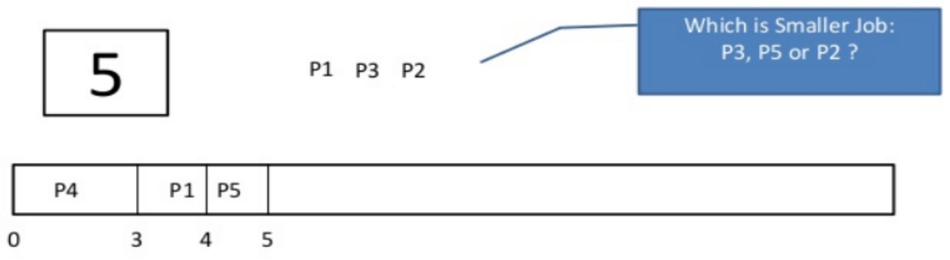
Step 4) At time = 4, process P5 will arrive. The burst time of P3, P5, and P1 is compared. Process P5 is executed because its burst time is lowest. Process P1 is preempted.

Process Queue	Burst time	Arrival time
P1	5 out of 6 is remaining	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

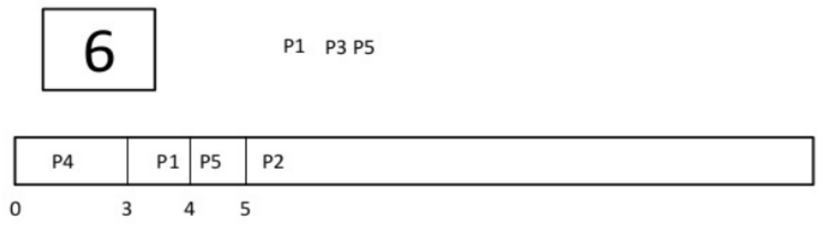


Step 5) At time = 5, process P2 will arrive. The burst time of P1, P2, P3, and P5 is compared. Process P2 is executed because its burst time is least. Process P5 is preempted.

Process Queue	Burst time	Arrival time
P1	5 out of 6 is remaining	2
P2	2	5
P3	8	1
P4	3	0
P5	3 out of 4 is remaining	4

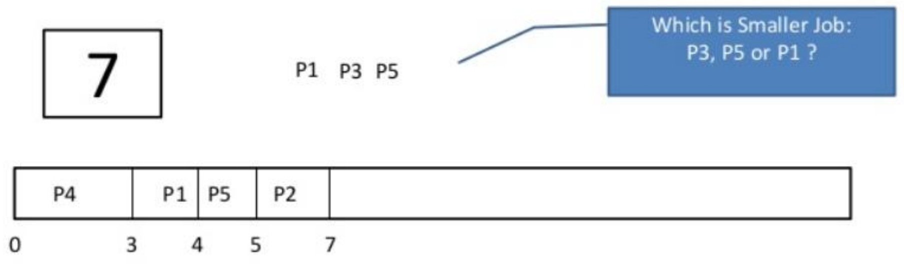


Step 6) At time =6, P2 is executing

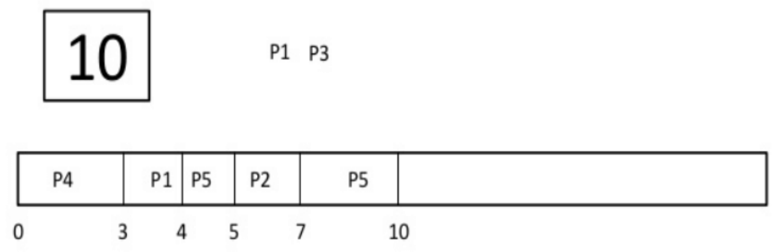


Step 7) At time =7, P2 finishes its execution. The burst time of P1, P3, and P5 is compared. Process P5 is executed because its burst time is lesser.

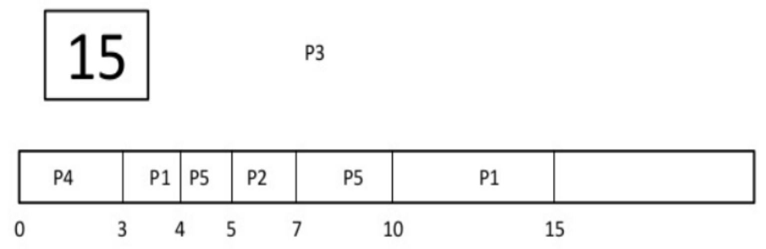
Process Queue	Burst time	Arrival time
P1	5 out of 6 is remaining	2
P2	2	5
P3	8	1
P4	3	0
P5	3 out of 4 is remaining	4



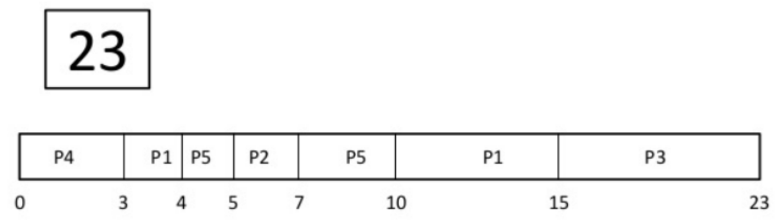
Step 8) At time =10, P5 will finish its execution. The burst time of P1 and P3 is compared. Process P1 is executed because its burst time is less.



Step 9) At time =15, P1 finishes its execution. P3 is the only process left. It will start execution.



Step 10) At time =23, P3 finishes its execution.



Waiting time
P4= 0-0=0
P1= (3-2) + 6 =7
P2= 5-5 = 0
P5= 4-4+2 =2
P3= 15-1 = 14

Average Waiting Time = 0+7+0+2+14/5 = 23/5 =4.6

2.3 Algorithm for non-preemptive SJF

Purpose: to minimize the turnaround time by executing the process with the shortest burst time first.

Input: 2D list A to store process information (Process Id, Burst Time, Waiting Time, Turnaround Time) for each process. Each row corresponds to a process, and each column corresponds to its different attributes.

Output: Average waiting time and average turnaround time. Display Gantt Chart.

Algorithm:

STEP 1: Initialization

- The n variable stores the number of processes to be scheduled.
- A 2D list **A** is created to store process information (**Process Id, Burst Time, Waiting Time, Turnaround Time**) for each process.

STEP 2: Input Burst Times

- The code prompts the user to enter the burst time for each process. The entered burst times are stored in the second column (**A[i][1]**) of the **A** list. The process IDs are also assigned incrementally (P1, P2, ...) to each process.

STEP 3: Sorting by Burst Time

- The processes are sorted based on their burst times using a simple selection sort algorithm.

STEP 4: Calculation of Waiting Times

- The waiting time for the first process is set to 0 (**A[0][2] = 0**).
- For each subsequent process, the waiting time is calculated by summing up the burst times of the processes that come before it. This represents the total time that the process has to wait before starting its execution.

STEP 5: Calculation of Turnaround Times

- The turnaround time for each process is calculated as the sum of its burst time and waiting time (**A[i][3] = A[i][1] + A[i][2]**).

STEP 6: Printing the Results

- The calculated waiting times, turnaround times, and burst times are printed for each process.
- The average waiting time and average turnaround time for all processes are computed and printed.

STEP 7: Gantt Chart Visualization

- The code creates a Gantt chart to visualize the scheduling timeline.
- The **gantt_pos** list is used to store the start and end positions of each process on the Gantt chart.
- The Gantt chart is created using a horizontal bar plot, where each process is represented by a colored bar.
- Process labels and starting times are annotated on the chart to show which process is executing at each point in time.

2.4 Algorithm for preemptive SJF

Algorithm schedulingProcess(process_data):

Purpose: Implement scheduling process using the Preemptive Shortest Job First (SJF) algorithm.

Input:

The function takes a list of process_data as input, where each element in the 2D list represents a process and contains the following information:

Index 0: Process ID

Index 1: Arrival Time (when the process arrives in the system)

Index 2: Burst Time (the time required for the process to complete its execution)

Index 3: Completion Status (0 means the process is not executed yet, and 1 means the process is completed)

Index 4: A copy of the burst time to display the output table

Each row of **process_data** corresponds to a process, and each column corresponds to the different attributes.

Output: Average waiting time and average turnaround time. Display Gantt Chart and sequence of processes.

Algorithm:

STEP 1: Initialization

- The **s_time** variable is used to keep track of the current time during the scheduling process.
- An empty list **sequence_of_process** is created to store the sequence in which the processes are scheduled.

STEP 2: Sorting the processes

- The **process_data** list is sorted based on the arrival time (index 1) of each process.

STEP 3: Main loop

- Inside the loop, two lists are created: **ready_queue** and **normal_queue**. These lists are used to store the processes that are ready to execute and those that are not ready yet (due to their arrival time).
- This portion runs indefinitely until all processes are completed (the loop will break when both **ready_queue** and **normal_queue** are empty).

STEP 4: Populating the queues

- Processes with an arrival time less than or equal to **s_time** and whose completion status is 0 (not executed yet) are added to the **ready_queue**.
- Processes with an arrival time greater than **s_time** and whose completion status is 0 are added to the **normal_queue**.

STEP 5: Executing processes from the ready queue

- If the **ready_queue** is not empty, it means there are processes that are ready to be executed.
- The **ready_queue** is sorted based on the burst time (index 2) of each process, and the process with the shortest burst time is selected for execution.
- The selected process' burst time is decremented by 1 to simulate its execution for one unit of time.
- If the burst time of the selected process becomes 0, it means the process has completed execution, so its completion status (index 3) is set to 1, and its completion time (index 5) is added to the process data.

STEP 6: Executing processes from the normal queue

- If the **ready_queue** is empty, it means there are no processes ready to be executed at the current time (**s_time**).
- In this case, the **normal_queue** is considered, and the process with the earliest arrival time is selected for execution (since there are no processes with arrival time less than or equal to **s_time**).
- The process execution and updating the completion status and completion time are done in the same way as for the processes in the **ready_queue**.

STEP 7: Calculating turnaround and waiting times

- After loop completes execution, the **calculateTurnaroundTime** and **calculateWaitingTime** methods are called to compute the turnaround time and waiting time for each process.

STEP 8: Printing the scheduling result

- The **printData** method is called to print the scheduling data, including the turnaround time and waiting time for each process.

STEP 9: Gantt chart visualization

- The **gantt_chart** and **gantt_chart2** methods are called to visualize the Gantt chart for the scheduled processes.

2.5 Environment for implementing SJF

Python has a rich ecosystem of libraries and frameworks that cover various domains, including mathematics, scientific computing, data analysis, and machine learning. These libraries provide pre-built functions and data structures that can significantly simplify the implementation of complex algorithms.

Matplotlib is a widely used Python library for creating static, interactive, and animated visualizations in a variety of formats. It's particularly useful for visualizing data, trends, and results from algorithms. Matplotlib provides a simple and intuitive interface for creating a wide range of visualizations. As such, we've used *matplotlib.pyplot.broken_barh*, which plots a horizontal sequence of rectangles to help visualize the gantt charts.

While Python's interpreted nature can make it slower than compiled languages like C++ for certain performance-critical tasks, the ease of development and the abundance of libraries often outweigh this drawback for algorithm implementation, as its dynamic nature and interpreted execution enabled rapid prototyping. This meant that we could quickly implement and test algorithms without the need for compilation, which sped up the development cycle and allowed for faster iterations.

Google Colab (short for Colaboratory) is a free cloud-based platform provided by Google that allows anybody to write and execute Python code in a Jupyter Notebook environment. It provides a range of useful features, making it a popular choice for various tasks, including algorithm implementation, data analysis, machine learning, and more. Google colab's multiple export features assisted us in saving Colab notebooks in various formats, including Jupyter.ipynb files, HTML, and PDF.

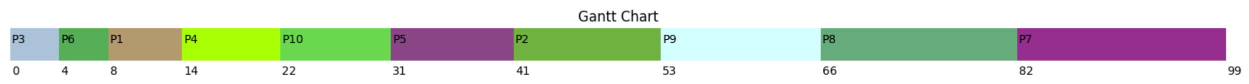
3. Results

3.1 Non-preemptive SJF:

The burst times of ten processes are given as input for our implementation of non-preemptive SJF program. The arrival times for each are not given as input and are instead considered 0; each having already arrived. The SJF algorithm selects the process with the shortest burst time to execute first, ensuring that the process with the least amount of work gets executed before longer processes.

Output:

```
Enter number of process: 10
Enter Burst Time:
P1: 6
P2: 12
P3: 4
P4: 8
P5: 10
P6: 4
P7: 17
P8: 16
P9: 13
P10: 9

Process ID        Burst Time        Waiting Time        Turnaround Time
P3                4                 0                   4
P6                4                 4                   8
P1                6                 8                   14
P4                8                 14                  22
P10               9                 22                  31
P5                10                31                  41
P2                12                41                  53
P9                13                53                  66
P8                16                66                  82
P7                17                82                  99

Average Waiting Time= 32.1
Average Turnaround Time= 42.0
```



After completing all processes, the average waiting time is calculated by summing up the waiting times of all processes and dividing by the number of processes (n = 10). Similarly, the average turnaround time is calculated by summing up the turnaround times of all processes and dividing by the number of processes.

Average Waiting Time: (0 + 4 + 8 + 14 + 22 + 31 + 41 + 53 + 66 + 82) / 10 = 32.1

Average Turnaround Time: (4 + 8 + 14 + 22 + 31 + 41 + 53 + 66 + 82 + 99) / 10 = 42.0

These averages represent the efficiency and performance of the scheduling algorithm in terms of minimizing waiting and turnaround times.

A Gantt chart has been used to visualize process scheduling. It is a graphical representation of a schedule, showing the start and finish times of various processes over a specific period. Each process is represented as a horizontal bar, and the length of the bar corresponds to the time it takes to complete the process.

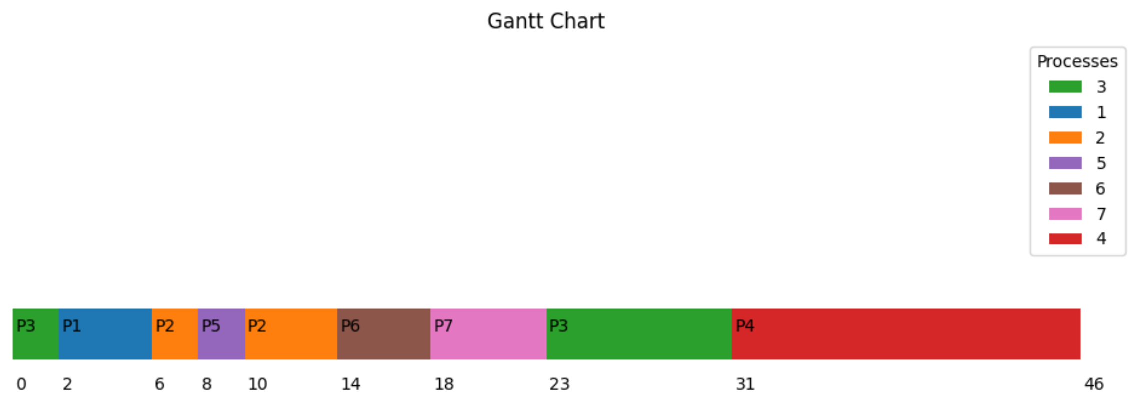
3.2 Preemptive SJF:

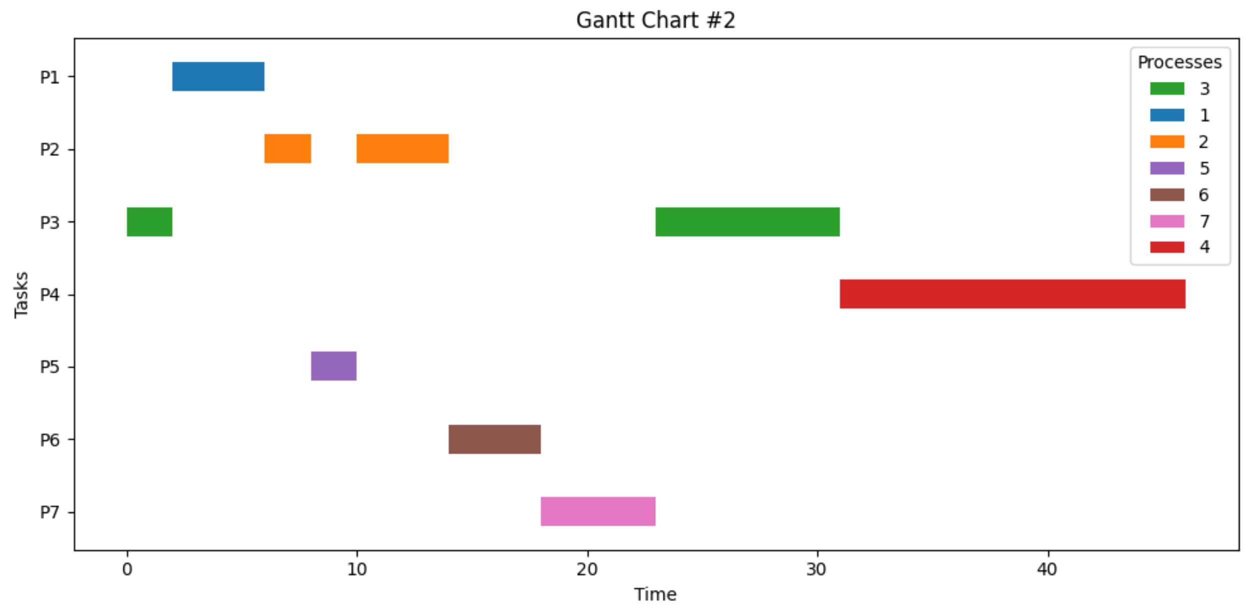
The following output represents the result of a preemptive Shortest Job First (SJF) scheduling algorithm applied to a set of 7 processes with their respective arrival times and burst times. In preemptive SJF, the process with the shortest remaining burst time is given priority.

The preemptive SJF algorithm selects the process with the shortest remaining burst time whenever a new process arrives or a running process completes its burst. It ensures that the process with the least remaining work gets executed next.

Output:

```
Enter number of processes: 7

P1
Enter Arrival Time: 2
Enter Burst Time: 4

P2
Enter Arrival Time: 4
Enter Burst Time: 6

P3
Enter Arrival Time: 0
Enter Burst Time: 10

P4
Enter Arrival Time: 7
Enter Burst Time: 15

P5
Enter Arrival Time: 8
Enter Burst Time: 2

P6
Enter Arrival Time: 10
Enter Burst Time: 4

P7
Enter Arrival Time: 12
Enter Burst Time: 5

Process ID        Arrival Time          Burst Time          Completion Time          Turnaround Time          Waiting Time
3                        0                   10                      31                       31                   21

1                        2                    4                       6                        4                    0

2                        4                    6                      14                       10                    4

4                        7                   15                      46                       39                   24

5                        8                    2                      10                        2                    0

6                       10                    4                      18                        8                    4

7                       12                    5                      23                       11                    6

Average Turnaround Time: 15.0
Average Waiting Time: 8.428571428571429
Sequence of Process: [3, 3, 1, 1, 1, 1, 2, 2, 5, 5, 2, 2, 2, 2, 6, 6, 6, 6, 7, 7, 7, 7, 7, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4,
4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]
```





The "Average Turnaround Time" and "Average Waiting Time" are calculated by summing up the respective times of all processes and then dividing both by the number of processes (n = 7).

Average Waiting Time: (21 + 0 + 4 + 24 + 0 + 4 + 6) / 7 = 8.428571428571429

Average Turnaround Time: (31 + 4 + 10 + 39 + 2 + 8 + 11) / 7 = 15.0

The "Sequence of Process" represents the order in which processes were executed based on the preemptive SJF algorithm. This sequence includes process IDs and shows how the CPU time was allocated to different processes in a preemptive manner. Two Gantt charts have also been used to visualize this.

4. Conclusion

It is evident that SJF offers significant benefits in terms of minimizing average waiting time and turnaround time. The algorithm's principle of prioritizing shorter jobs first leads to efficient resource utilization and improved system performance. However, the non-preemptive nature of SJF can result in longer jobs experiencing starvation, which can be addressed by using preemptive SJF or incorporating aging techniques. It's essential to note that the effectiveness of SJF heavily relies on accurate job length predictions, which might not always be feasible in dynamic and unpredictable environments. Additionally, SJF's fairness could be questioned due to its bias towards shorter jobs, potentially leading to longer jobs being consistently delayed.

In scenarios where job lengths are known or can be estimated reliably, SJF can be a valuable scheduling choice. Yet, for systems with diverse job lengths and changing priorities, a hybrid approach that combines SJF with other scheduling algorithms might provide a more balanced and adaptable solution. Ultimately, the choice of scheduling algorithm, including SJF, should be based on a thorough understanding of the system's characteristics, requirements, and the trade-offs associated with different scheduling strategies.

6. References

- Tri Dharma Putra. "Analysis of Preemptive Shortest Job First (SJF) Algorithm in CPU Scheduling." International Journal of Advanced Research in Computer and Communication Engineering Vol. 9, Issue 4, April 2020

- Silberschatz, Galvin and Gagne. "Operating System Concepts – 10th Edition." Chapter 5: CPU Scheduling.

- Lawrence Williams. "Shortest Job First (SJF): Preemptive, Non-Preemptive Example." July 24, 2023. https://www.guru99.com/shortest-job-first-sjf-scheduling.html

- Python documentation: https://docs.python.org/3/library/functions.html

- Matplotlib documentation: https://matplotlib.org/3.5.3/api/_as_gen/matplotlib.pyplot.html