
CSD-223 Data Structures

Project Report



Submitted To:

Er. Nitin Gupta

Submitted By:

Group Leader

Anshul 185003

Email :- 185003@nith.ac.in

Phone no. :- 8544759631

Group Members

Ananya Sharma 185001

Email :- 185001@nith.ac.in

Phone no. :- 8580415318

Jyoti Sharma 185022

Email :- 185022@nith.ac.in

Phone no. :- 8091790670

Problem and its Description

CPU Scheduling algorithms are one of the most used algorithms based on which our Operating Systems evolved over time . For a better functioning of our Operating Systems , they need to be as efficient as possible . For studying them we mostly refer to the internet where we find the likes of websites such as geeksforgeeks , tutorialspoint etc. where majority of these algorithms are available in $O(n^2)$ complexity and a very few in $O(n \log n)$ time complexity.

Now , the question is what is Time Complexity ? The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution. In simple words, every piece of code we write, takes time to execute. The time taken by any piece of code to run is known as the time complexity of that code. The lesser the time complexity, the faster the execution .

Lets take an example to see why time complexity is important ? Lets say there are two algorithms which are doing the same task , first one having a higher time complexity of $O(n^2)$ but the second one is having a lower time complexity of $O(n \log n)$. There is a computer which can execute 10^8 instructions in a second . If there is a task which is to be performed by the above two algorithms and in this task 10^8 instructions are to be performed , the first algorithm will be executed in 10^8 seconds and the second algorithm having a time complexity of $O(n \log n)$ will be executed in approx 26 seconds . So , by this we can understand how important it is to make an algorithm which has a lower time complexity

The other problem encountered is that all these algorithms are implemented using different data structures . As every algorithm is using different data structure , despite the objective of all algorithm is same , it creates dissimilarity in terms of understanding and time complexity . If all these are implemented using same data structure , they will have some sort of similarity and at the same time it becomes easy for understanding . And all the algorithms can be made efficient by just working on one algorithm and applying the same for the rest .

So , the choice of data structure becomes equally important as data structure will decide which algorithm to apply which will then decide the time complexity .

So , we are trying to design CPU scheduling algorithms in such a way so that they have a low time complexity and all of them should be implemented by the same data structure in order to maintain the similarity as objective of all the algorithms are same . At the same time , we are making a GUI for its better implementation and so that every one can use this as a simulator .

Approach

We are trying to design CPU scheduling algorithms in such a way so that they have a low time complexity and all of them should be implemented by the same data structure in order to maintain the similarity as objective of all the algorithms are same . There are total 6 algorithms which we are going to implement . Here is a brief introduction about all these 6 algorithms. :-

1. **FCFS** :- First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue.
2. **SJF(Non-Preemptive)** :- Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-preemptive algorithm.
3. **SJF(Preemptive)** :- In this , the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.
4. **Priority(Non-Preemptive)** :- In the Non Preemptive Priority scheduling, The Processes are scheduled according to the priority number assigned to them. Once the process gets scheduled, it will run till the completion.
5. **Priority(Preemptive)** :- In Preemptive Priority Scheduling, at the time of arrival of a process in the ready queue, its Priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time. The One with the highest priority among all the available processes will be given the CPU next.
6. **Round Robin** :- Round Robin is the preemptive process scheduling algorithm.Each process is provided a fix time to execute, it is called a quantum.Once a process is executed for a given time period, it is preempted and other process executes for a given time period.

All these algorithms are classified into two main categories :-

1. **Preemptive** :- Preemptive scheduling is used when a process switches from running state to ready state or from waiting state to ready state. The resources (mainly CPU cycles) are allocated to the process for the limited amount of time and then is taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in ready queue till it gets next chance to execute.

Approach continued...

2. **Non-Preemptive** :- Non-preemptive Scheduling is used when a process terminates, or a process switches from running to waiting state. In this scheduling, once the resources (CPU cycles) is allocated to a process, the process holds the CPU till it gets terminated or it reaches a waiting state. In case of non-preemptive scheduling does not interrupt a process running CPU in middle of the execution. Instead, it waits till the process complete its CPU burst time and then it can allocate the CPU to another process.

Now , the main task is of choosing the accurate data structure . The data structure to be chosen depends upon the type of data and what kind of algorithms are to be performed .

In CPU scheduling algorithms , there are 3 common inputs in all of the algorithms , brief about these are given below :-

1. **Process Id** :- Process id the unique id given to every process . This helps in identification of a particular process .
2. **Arrival Time** :- It is the time at which a particular process arrives in the ready queue .
3. **Burst Time** :- Burst time is the amount of time required by a process for executing on CPU. It is also called as execution time or running time.

Priority algorithms require priority as an extra input whereas round robin algorithm require time quantum as an extra input .

The objective of all the algorithms are same . In all these algorithms we have to find :-

1. **Completion Time** :- It is the time at which the process completes its execution .
2. **Turnaround Time** :- Turnaround time (TAT) is the time interval from the time of submission of a process to the time of the completion of the process. It can also be considered as the sum of the time periods spent waiting to get into memory or ready queue, execution on CPU and executing input/output.
3. **Waiting Time** :- Waiting time is the total time spent by the process in the ready state waiting for CPU.
4. **Average Waiting Time** :- It is the average of waiting times of all the processes .
5. **Average Turnaround Time** :- It is the average of turnaround times of all the processes.

Approach continued...

As objective of all these algorithms are same , so to maintain a similarity between all the algorithms , same data structure should be used for all the algorithms . Now , the question is why to use same data structure for all the algorithms ? By using the same data structure it becomes easy to understand the algorithms as one only need to understand the workflow of one algorithm and the rest can be understood very easily . It becomes easy for debugging .

Now , the next question that arrives is on what basis the data structure is to be chosen? It depends upon the type of operations that are to be performed and the type of data that is going to be input . The data structure chosen , will decide the algorithms to be applied which will then decide the time complexity . Time complexity is the most crucial factor for deciding an algorithm . So , it can be said that choosing a right data structure is very important .

Now , let us know about the operations that are to be performed . The main operations that are to be performed are sorting , addition , deletion , shuffling after every time quantum or after any new process arrive or after any process is completed . Now , a data structure which supports efficient insertion and deletion and also the sorting should be very time efficient should be chosen .As there are a lot of operations to be performed after every time quantum or after any new process arrive or after any process is completed , it becomes very difficult to reduce the time complexity and each process has a process id , arrival time , burst time , completion time ,so it becomes even more difficult to do all the operations simultaneously on all these while keeping the time complexity as low as possible .

After a lot of research and trials , the most suitable data structure found is heap , using which sorting , insertion and deletion becomes very time efficient . It supports sorting in $O(n \log n)$ time and insertion and deletion in $O(\log n)$ time .

Let us start with a brief introduction about heap . A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

1. **Max Heap** :- In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
2. **Min Heap** :-In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

Approach continued...

Now, it becomes extremely difficult for us to break down each algorithm and redesign them using heap as heap implementation is little difficult, and in this we have to implement these 6 complex algorithms which require sorting, addition, deletion, shuffling after every time quantum or after any new process arrive or after any process is completed, it becomes even more difficult.

The first task to be accomplished is sorting which is the common task for every algorithm, so a function is made for it. As user inputs the arrival time and the burst time for all the processes, the processes should be sorted according to the arrival time. So, we have to sort on the basis of arrival time and simultaneously sorting the burst time, process id (even the priority in priority algorithms) in the same order as arrival time and that too in $O(n \log n)$. So, it becomes little difficult.

A new array is taken which becomes the heap for sorting. The size of this array is 1 greater than other array sizes as when heap is implemented using array, the array is used from the 1st index. The arrival time is copied into that new array by forming a max heap. While any swap operation is applied eg:- i^{th} index value is swapped with j^{th} index value, then in the remaining arrays eg burst time, process id, the swap operations are applied $(i - 1)^{th}$ index value is swapped with $(j - 1)^{th}$ index value because in heap indexes are from 1 to n and in rest index are from 0 to $n - 1$. Now, heap sort is applied on this max heap and the swapping process remains the same as in making of heap sort. So, by this algorithm on the basis of arrival time all the arrays(process id, burst time, priority) are sorted using heap and that too in $O(n \log n)$ time complexity, which is considered as one of the most efficient time complexities.

There are some sorting techniques which can sort in $O(n)$ time, but almost none of these can be applied here as they come with some constraints, some with constraints that all the values should be different and some with constraints that values should not be greater than 10^6 as they require making of a new array where the index corresponds with the value and array of size greater than 10^6 is not possible. So, in this case $O(n \log n)$ is considered one of the best.

The first task of sorting is accomplished and that too in $O(n \log n)$ time complexity. Now, the next task is to break down each algorithm and then redesigning them so that they are implemented using heap.

Approach continued...

FCFS

First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue. It is the simplest algorithm to implement. After the values getting sorted, we have to process the processes in the same order, so nothing special to implemented is required. It is implemented in $O(n)$.

```
470 #FCFS
471 def FCFS(processes, burst_time, arrival_time):
472     n = len(processes)
473     completion_time = [0]*n
474     curr_time = 0
475     for i in range(n):
476         #Checking whether the process has yet arrived or not
477         #If not arrived, then increasing the current time to make the process arrive
478         if(curr_time < arrival_time[i]):
479             curr_time = arrival_time[i]
480         #now calculating the completion time
481         curr_time = curr_time + burst_time[i]
482         completion_time[processes[i] - 1] = curr_time
483     return completion_time
```

SJF Non Preemptive

Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJF is a non-preemptive algorithm. In this a variable is used for depicting current time and it is initialized with 0, and a heap is used for sorting burst time. The thing that is inserted in heap is burst time. After the sort function is applied, the first process is inserted in to the min heap and the current time is made equal to the arrival time of that process. Then all the processes having arrival time less than or equal to the current time are inserted into the min heap. After inserting, as the first index of min heap contains the smallest element, and in this case it contains the process with smallest burst time, so it is removed from the heap and its burst time is added into the current time, which means that it has executed. Now, all the processes having arrival time less than or equal to the current time are inserted in the min heap. Now again the value in the first index is deleted and added to the current time. This process of inserting into heap and deleting the first value continues till all values of burst time are inserted into heap. When all the process are inserted into the min heap, one by one process is deleted, as in min heap, we can only delete the minimum value, so the deletion is done in minimum to maximum order. After deletion the burst time is added to the current time. A process id heap is also maintained which contains the id of processes in heap. The operations that are applied on process id heap are same as operations applied on burst time heap. Another array of completion time is also created, in which insertion is done after every process is removed from heap. The process id heap acts as a hashtable for its insertion. The reason for doing such insertion is to keep the completion time array sorted (in terms of process id). So, here we used heap and hashtable for making this algorithm time efficient. It has $O(n \log n)$ time complexity.

Approach continued...

SJF Preemptive

In this , the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time. It is a preemptive algorithm. In this a variable is used for depicting current time and it is initialized with 0 , and a heap is used for sorting burst time . The thing that is inserted in heap is burst time. After the sort function is applied , the first process is inserted in to the min heap and the current time is made equal to the arrival time of that process . Then all the processes having arrival time less than or equal to the current time are inserted into the min heap . Another variable is used for storing the next time , because in preemptive algorithms a process which is running , it runs until the next process arrives and then the conditions are checked again . The next time is storing the arrival time of next process . When all the processes having arrival time less than or equal to current time are inserted in min heap , the first process runs till the time in next time variable and the value (next time - current time) is decreased from the first index value , which denotes that the process having minimum burst time runs for (next time - current time) time . But if (next time - current time) is greater than the process having minimum burst time which is the process in first index of heap , then it will be executed completely , which means it is deleted from heap . The next time and current time are incremented accordingly . This process of inserting into heap and deleting the first value continues till all values of burst time are inserted into heap . When all the process are inserted into the min heap , one by one process is deleted , as in min heap , we can only delete the minimum value , so the deletion is done in minimum to maximum order . After deletion of each process the burst time is added to the current time . A process id heap is also maintained which contains the id of processes in heap . The operations that are applied on process id heap are same as operations applied on burst time heap . Another array of completion time is also created , in which insertion is done after every process is removed from heap . The process id heap acts as a hashtable for its insertion . The reason for doing such insertion is to keep the completion time array sorted (in terms of process id) . So , here we used heap and hashtable for making this algorithm time efficient . It has $O(n \log n)$ time complexity.

Priority Non-Preemptive:-

In the Non Preemptive Priority scheduling, The Processes are scheduled according to the priority number assigned to them. Once the process gets scheduled, it will run till the completion. It is a non preemptive algorithm . In this , priority is inserted into heap . After the sort function is applied , the first process is inserted in to the max heap and the current time is made equal to the arrival time of that process . Then all the processes having arrival time less than or equal to the current time are inserted into the max heap .

Approach continued...

After all the processes having arrival time less than or equal to current time is inserted into the max heap , the highest priority process is to be selected and the first element of the max heap is maximum , so it is deleted from heap , which means it gets executed . Its burst time is added to the current time . Now again the process of inserting into heap and deleting continues till all the processes are added into the heap . When all the processes are added into the max heap , processes get executed one by one according to higher to lower priority . So , processes gets deleted one by one from the heap . As process having highest priority is at the first index of the heap , so it gets deleted . After deletion the burst time is added to the current time . A process id heap is also maintained which contains the id of processes in heap . The operations that are applied on process id heap are same as operations applied on priority heap . Another array of completion time is also created , in which insertion is done after every process is removed from heap . The process id heap acts as a hashtable for its insertion . The reason for doing such insertion is to keep the completion time array sorted(in terms of process id) . So , here we used heap and hashtable for making this algorithm time efficient . It has $O(n \log n)$ time complexity.

Priority Preemptive:-

In Preemptive Priority Scheduling, at the time of arrival of a process in the ready queue, its Priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time. The One with the highest priority among all the available processes will be given the CPU next. In this , priority is inserted into heap . After the sort function is applied , the first process is inserted in to the max heap and the current time is made equal to the arrival time of that process . Then all the processes having arrival time less than or equal to the current time are inserted into the max heap . Another variable is used for storing the next time , because in preemptive algorithms a process which is running , it runs until the next process arrives and then the conditions are checked again . The next time is storing the arrival time of next process . When all the processes having arrival time less than or equal to current time are inserted in max heap , the first process runs till the time in next time variable and the value (next time - current time) is decreased from the its burst time , which denotes that the process having maximum priority runs for (next time - current time) time . But if (next time - current time) is greater than the burst time of process having maximum priority , then it will be executed completely , which means it is deleted from heap . The next time and current time are incremented accordingly . This process of inserting into heap and deleting the first value continues till all values of burst time are inserted into heap . When all the processes are added into the max heap , processes get executed one by one according to higher to lower priority . So , processes gets deleted one by one from the heap . As process having highest priority is at the first index of the heap , so its get deleted . After deletion the burst time is added to the current time . A process id heap is also maintained which contains the id of processes in heap . The operations that are applied on process id heap are same as operations applied on priority heap . Another array of completion time is also created , in which insertion is done after every process is removed from heap .

Approach continued...

The process id heap acts as a hashtable for its insertion . The reason for doing such insertion is to keep the completion time array sorted(in terms of process id) . So , here we used heap and hashtable for making this algorithm time efficient . It has $O(n \log n)$ time complexity.

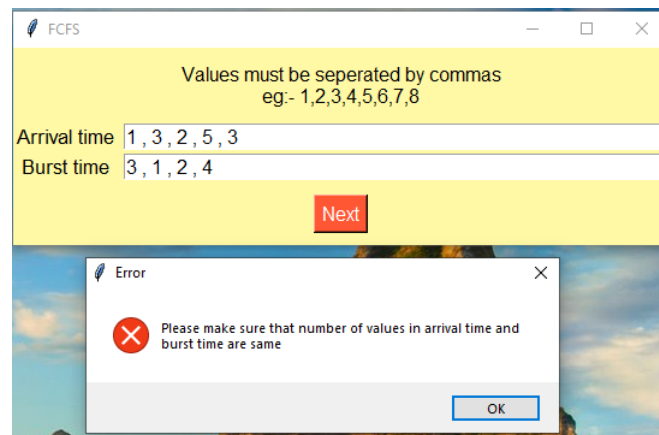
Round Robin:-

Round Robin is the preemptive process scheduling algorithm. Each process is provided a fix time to execute, it is called a quantum. Once a process is executed for a given time period, it is preempted and other process executes for a given time period. This algorithm is implemented with the help of a queue . Burst time is entered into the queue . The first process is entered into the queue and the current time is made equal to its arrival time . Then all the processes having arrival time less than or equal to current time is inserted into the queue . Another variable is used to store the time quantum which is input by the user . The first process in the queue is dequeued and checked whether it is greater than time quantum or not . If the burst time of first process is less than or equal to the time quantum , then it is executed completely . Current time is incremented accordingly . Then all the processes having arrival time less than or equal to current time is inserted into the queue . If time quantum is less than the burst time , then the process is executed for time quantum time and current time is incremented accordingly . Then all the processes having arrival time less than or equal to current time is inserted into the queue and time quantum is subtracted from the burst time of dequeued process . It is then inserted in the queue. This process continues till all the processes are inserted into the queue . After all the process are inserted , first process in the queue is dequeued and checked whether it is greater than time quantum or not . If the burst time of first process is less than or equal to the time quantum , then it is executed completely . Current time is incremented accordingly . If time quantum is less than the burst time , then the process is executed for time quantum time and current time is incremented accordingly . Time quantum is subtracted from the burst time of dequeued process . It is then inserted in the queue. This process continues till the queue becomes empty . A process id queue is also maintained which contains the id of processes in queue . The operations that are applied on process id queue are same as operations applied on burst time queue . Another array of completion time is also created , in which insertion is done after every process is executed completely . The process id queue acts as a hashtable for its insertion . The reason for doing such insertion is to keep the completion time array sorted(in terms of process id) . So , here we used queue and hashtable for making this algorithm time efficient . The queue used in round robin is a circular queue because in circular queue enqueue and dequeue are both done in $O(1)$.

Approach continued...

After implementing all these algorithms , next task was to implement functions for calculating turnaround time , waiting time , average waiting time . All these algorithms are implemented using 1 for loop which means they have $O(n)$ time complexity . After this , the task was to plot graphs which is plotted using the matplotlib library .

Now , we don't wanted to limit it to a console based simulator , we wanted to make this simulator in such a way so that the persons who are not familiar with the console based programming can also use it . So , we thought of giving it a GUI , which is implemented using tkinter . Implementing GUI was not an easy task because we have to make it simple and at the same time it should look good . The GUI made should be interactive and must tell people about almost all the errors in an understandable way . In a future update or change , the error displaying will be made better .



As seen in the above image , if someone enters different number of values in arrival time(5 in above image) and burst time(4 in above image) . The size of arrival time and burst time should be same and as seen in above image size of arrival time(size is 5) and burst time(size is 4) is different , so it gives an error .

Language Used and Why Used?

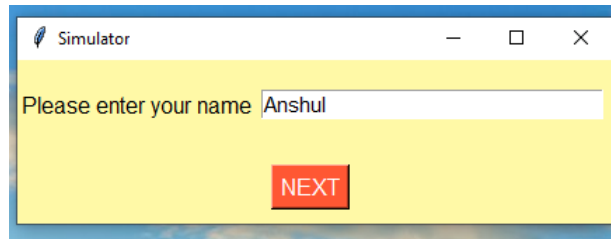
The language that we used to build this project is python . The reason for using python is because we wanted to plot the graphs using matplotlib and implement the GUI using tkinter .

Everything is made from scratch in this project . Circular queue is made from scratch . Heap sort and heap is implemented from scratch . All the functions are implemented from scratch. All the algorithms are implemented from scratch . The only places where the libraries are used are for plotting graphs and making the GUI .

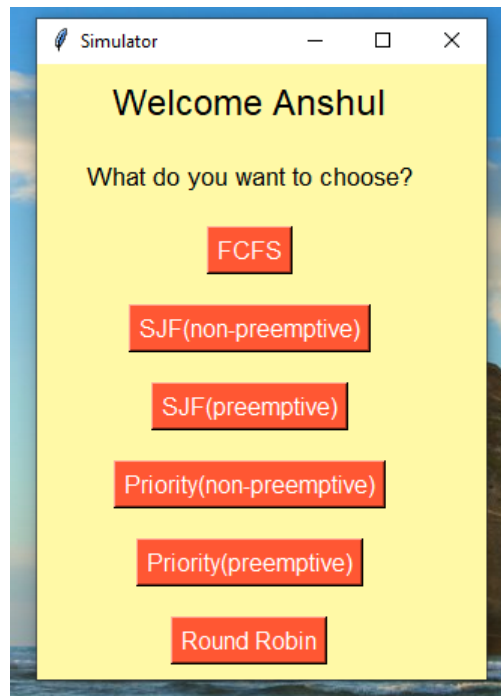
Key Points

1. It is a GUI based simulator.
2. Everything is implemented from scratch
3. It is very time efficient . It follows $O(n \log n)$ time complexity provided it includes sorting , adding , deleting and shuffling of processes after every new process arrives , after every process ends and after every time quantum .
4. It uses data structures like circular queue , hashtable , heap to make algorithms efficient.
5. It uses heap sort to make sorting faster which follows $O(n \log n)$ time complexity .
6. Algorithms are implemented in such a way that there is some sort of similarity between all the algorithms which makes it easy to understand all the algorithms.
7. It also plots graphs , so that one can visualize the results .
8. It can be used in labs and can be used by the upcoming students .
9. It can be used by the teachers for teaching CPU scheduling topic by using real time graphs which this simulator plots .
10. This simulator has been converted to an .exe application which means it can run on any windows computer and you don't have to install any other packages to run this simulator.

Snapshots

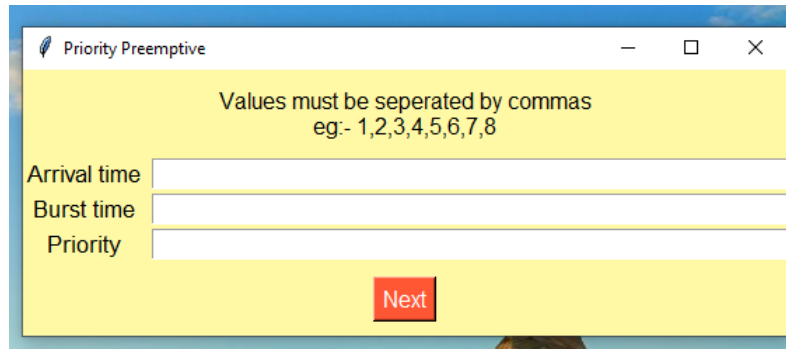


Firstly you will be asked to enter your name.



Then you can choose any of the algorithms to implement.

Snapshots



Priority Preemptive

Values must be seperated by commas
eg:- 1,2,3,4,5,6,7,8

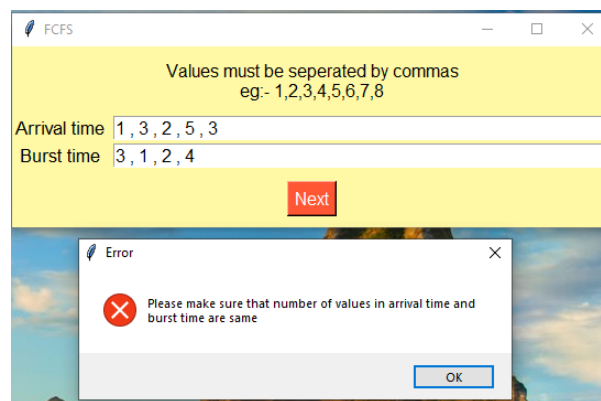
Arrival time

Burst time

Priority

Next

Then you will fill the details.



FCFS

Values must be seperated by commas
eg:- 1,2,3,4,5,6,7,8

Arrival time

Burst time

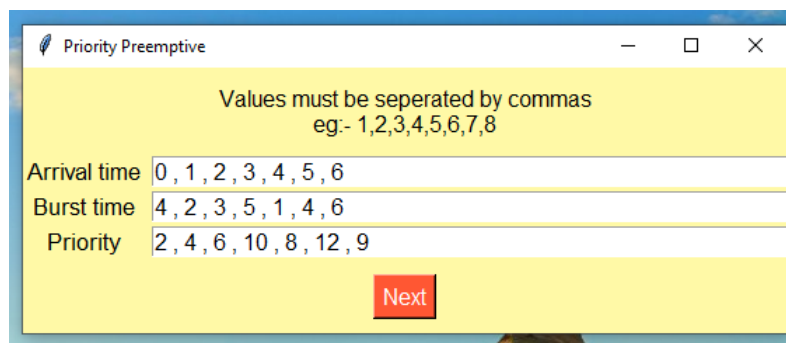
Next

Error

Please make sure that number of values in arrival time and burst time are same

OK

You will get notified about the error if you fill incorrect values .



Priority Preemptive

Values must be seperated by commas
eg:- 1,2,3,4,5,6,7,8

Arrival time

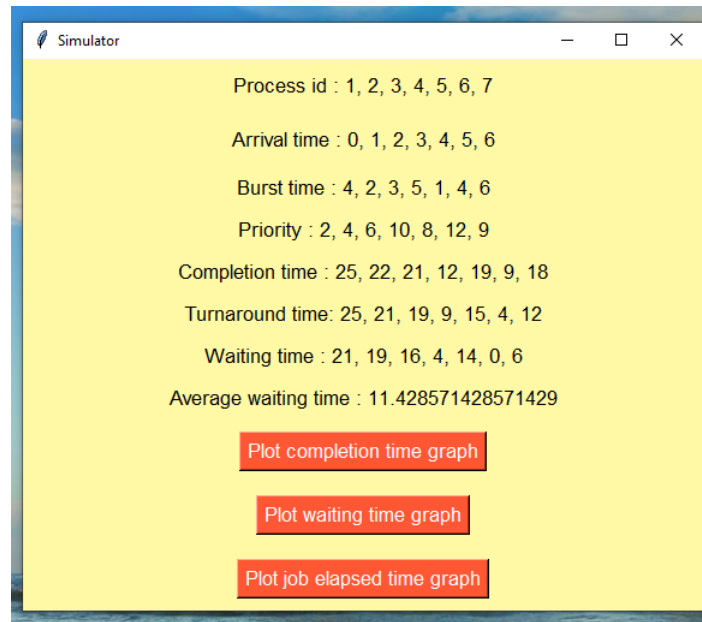
Burst time

Priority

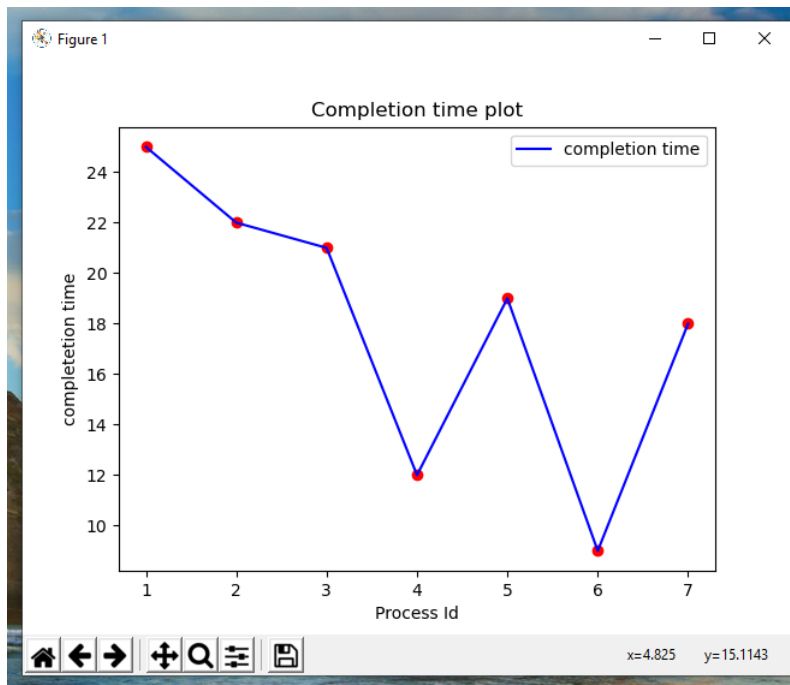
Next

After filling the correct details , click next.

Snapshots

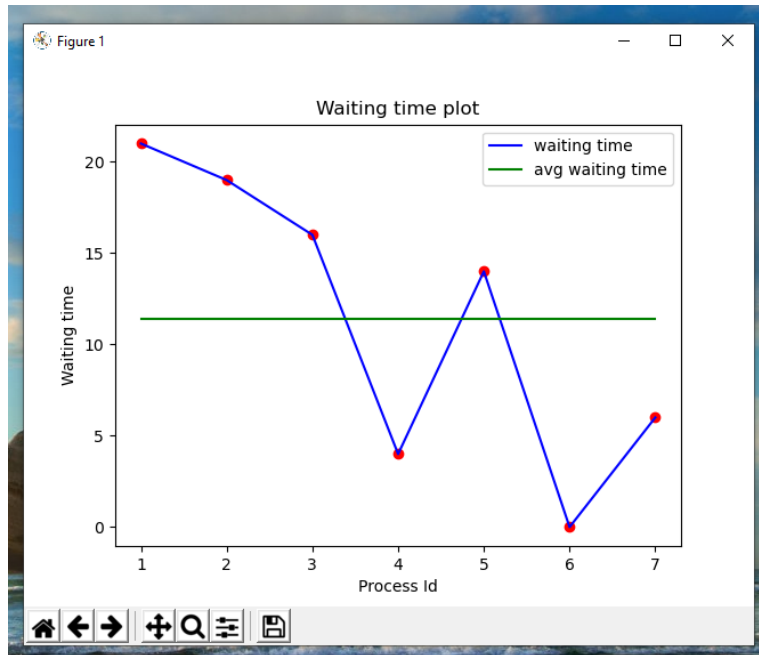


You will get results and you can choose to plot any graph .

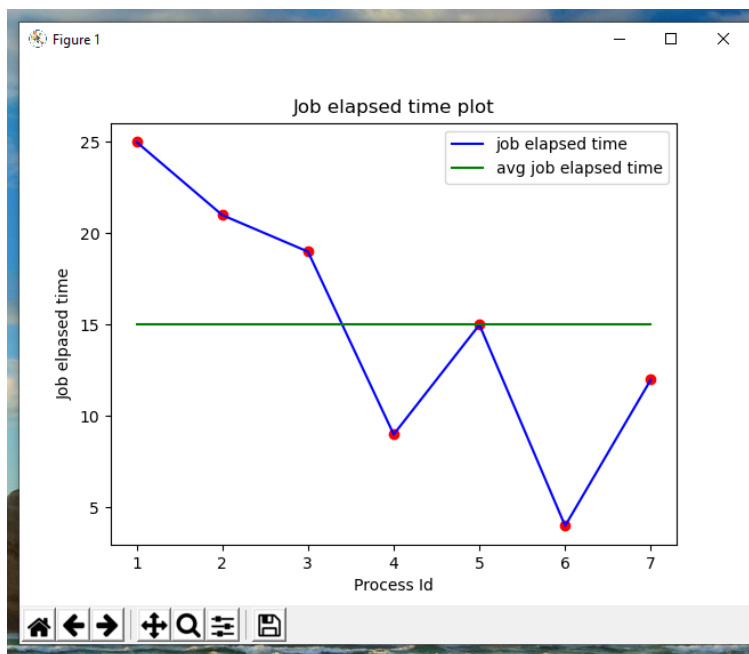


This is the completion time graph.

Snapshots



This is the waiting time graph.



This is the job elapsed time graph.