7SENG012W
Software Development Environments
Assignment 1

Caner Sakallilar, W1550786
Instructor: **Dr. George Charalambous**
Date of Submission: 01/12/2022

Abstract - The research undertaken within this report will lead to a script fulfilling all aspects of a complex specification for emulating a RR algorithm with provided input. The report demonstrates the functionality and implementation of three short-term scheduling algorithms, namely; FCFS, SJF, and RR. Examining their influence in practice can be challenging and time-consuming due to the necessity of modifying their test operating system kernel code and measuring the ensuing performance under a constant workload of actual applications. As the processor is the most significant resource, CPU scheduling becomes crucial for achieving operating system (OS) design objectives. The research provides a sound background for the rest of the report when developing the design of the script to emulate a RR algorithm from the specification of the system set by the client. This makes it much simpler to comprehend what is occurring within the system and why different sets of processes are candidates for CPU allocation at different instances of time. The purpose of this report is to examine the highly efficient CPU scheduler based on the construction of a high-quality scheduling algorithm that meets client objectives and to present evidence of a high functioning script with further evidence of testing.

Contents:

1. Critically Evaluate and Clear Documentation of Different Scheduling Algorithms

1.1 Introduction

The purpose of this paper is to compare and evaluate three short-term scheduling algorithms. More specifically: FCFS (First Come First Served), SJF (Shortest Job First), and RR (Round Robin).

Scheduling is a fundamental aspect of every operating system since it allows all computer resources to be planned before they are used, making the CPU arguably the most important computer resource. As a result, its scheduling algorithm is a critical component of the OS's architecture. When many processes are executable, it is up to the operating system to select which one should be executed first. The portion of the operating system that makes this choice is known as the scheduler, and the algorithm that it employs is known as the scheduling algorithm.

1.2 Preemptive vs Non-Preemptive Scheduling

A computer contains numerous active processes. Whenever a process executes, the CPU is assigned to that process. Occasionally, it is important to halt the current process' execution and grant precedence to another. Mechanisms for process scheduling might be preemptive or nonpreemptive. The following provides a brief summary to illustrate the distinction between Preemptive and Non-preemptive Scheduling in OS. Preemptive scheduling is the method for process scheduling that allows the execution of a process to be halted by another process, whilst, non-preemptive scheduling is the technique for process scheduling in which a process begins execution only after the execution of the preceding process has concluded.

1.3 Scheduling Algorithm Criteria

Multiple criteria are used to evaluate the efficiency of different scheduling algorithms. CPU scheduling algorithms have varying features, and the selection of a specific algorithm may favour one category of operations over another. In order to determine which algorithm to employ in a given circumstance, it is necessary to evaluate the qualities of each algorithm. This can be determined by the following criteria:

1. Utilization of CPU
2. Burst Time
3. Arrival Time
4. Completion Time
5. Waiting Time
6. Response Time
7. Throughput
8. Turnaround Time
9. Fairness

See 'Appendix A' for further explanation of each criteria and their respective calculation.

1.4 First Come First Served

FCFS scheduling (also known as FIFO - First In First Out) is an example of non-preemptive scheduling. The CPU is allotted to the process that wants the CPU first. If many processes arrive at the same moment, they are executed in the order they appear in the queue. The procedure in the back of the line must wait until all processes in front have been completed. Any process that is currently in the ready state is added to the FCFS queue based on its arrival time. The process no longer enters the queue upon completion. In this case, however, the process with the short burst time must wait for the preceding process to complete.

1.5 Shortest Job First

SJF can either be non-preemptive or preemptive, the latter commonly known as Shortest Remaining Time First, SRTF. However, for purposes of this report, only the non-preemptive version of this algorithm will be discussed.

SJF is utilised for reducing response time, resulting in minimal average process waiting time as well as turnaround time. Thus the total time from a process's arrival to its conclusion. Waiting time in SJF refers to the time a process spends whilst ready to execute in the queue. The average wait time is reduced due to the execution of processes with less burst time regardless of importance.

1.6 Round Robin

This method was developed specifically as a system of time-sharing. Typically, the RR approach is implemented in multiprogramming systems where CPU multitasking would be beneficial. Each system is allocated a specific amount of time, and so RR aims to fairly divide computer resources amongst the processes.

1.7 Example Data File

For the purpose of this report, the three aforementioned algorithms will be compared in terms of their average turnaround times, average waiting times, and average response times. The data file will be the same for all algorithms and will consist of five processes with varying arrival times and burst times. The round robin algorithm will also have a time quanta value of one. It is worth noting that results for this report will be presented in milliseconds (m/s). The data file is as follows:

Process	Arrival Time (AT)	Burst Time (BT)
P1	3	1
P2	0	1
P3	2	5
P4	4	2
P5	2	4

Table 1: Data file (example)

1.8 Comparison of the Algorithms in OS

See 'Appendix B' for a full breakdown of Quantitative Results.

The simplest algorithm to implement in OS is the FCFS, this being said however, it has the potential of causing a 'convoy effect'. When this occurs, "all the jobs queued behind it must wait a long time for the long job to finish."(Behzad, S. Et al, 2013) resulting in the slowing down of the whole OS.

When compared to the FCFS scheduling method, the SJF method is superior since it takes into consideration the amount of time required to finish a procedure (CPU burst) negating the potential convoy effect. It is arguable that SJF is the better option of the three, since it has, in this example, both the smallest average turnaround time (5.2) as well as the smallest average waiting time (2.6) respective to

FCFS (average turnaround time - 6.8, average waiting time - 4.2), and RR (average turnaround time - 7.2, average waiting time - 4.6). (*See 'Fig. 1' and 'Fig. 2' below)*.

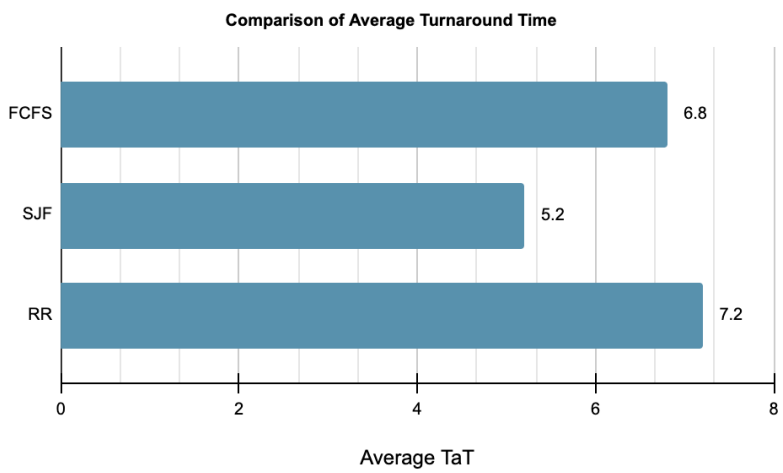


Fig. 1: Comparison of Average Turnaround Time

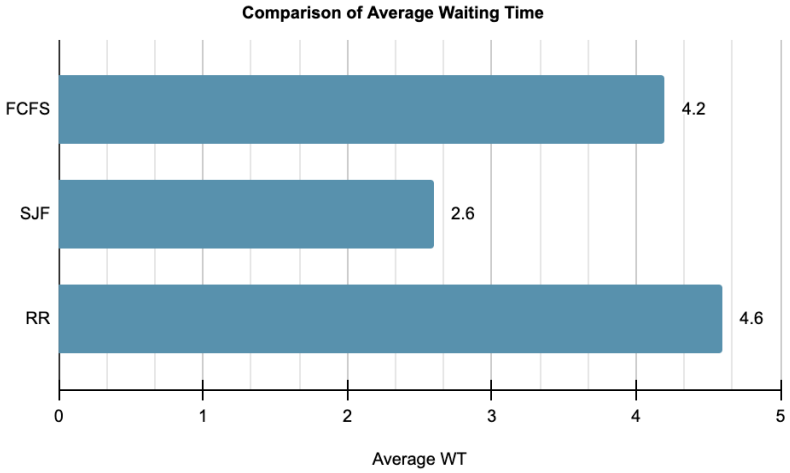


Fig. 2: Comparison of Average Waiting Time

However, since it is impossible to accurately forecast how long the subsequent CPU burst would last, implementing SJF scheduling in the OS may be challenging. In addition, an effect known as 'starvation' could occur. Starvation being "a problem encountered in multitasking where a process is perpetually denied necessary resources. Without those resources, the program can never finish its task." (Adekunle, Y.A. Et al, 2014) therefore, in contrast to FCFS, with SJF processes that have higher running durations may perish if there is an excessive number of processes with shorter run times.

A way to combat these issues could be via implementation of a RR algorithm. An RR algorithm promotes a more 'fair' approach to multitasking in OS as it gives an equal share of a 'time-slice' to each process. Although average times mentioned above are higher, in this example, whilst eliminating the potential issues caused from the other two algorithms, RR also has the quickest response time (see '*Fig. 3*' below). With RR average response time being 1.2, a considerable improvement on FCFS (4.2) and more than twice as quick as SJF (2.6).

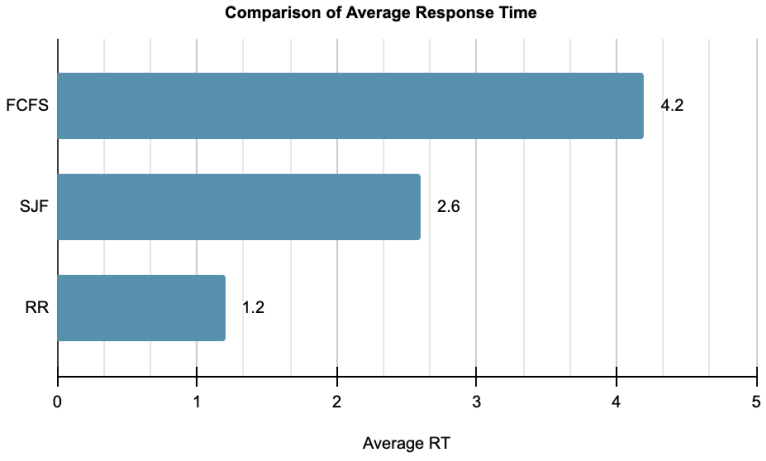


Fig. 3: Comparison of Average Response Time

Therefore, it can be argued that a solution to the problem of unfairness is provided by RR in the form of an improved average response time. However, this almost always results in a longer average time for the other characteristics being compared as is the case with the example presented in this report.

Ultimately, there is a trade-off between the first two aspects compared (average turnaround time and average waiting time) and the third aspect (average response time). As FCFS and SJF are nonpreemptive they will not terminate a process until it completely finishes its respective task at hand. This can cause convoy effects and starvation, whereas an RR algorithm bypasses these issues so long as the time quanta is neither too small nor too large. If the time quanta is too small then the algorithm becomes FCFS, if too small then context switching occurs leading to a large overhead and inefficiency of the OS.

1.9 Conclusion

In this report, three short-term scheduling algorithms have been presented with an example case study to show quantitative comparisons. Respective Gantt charts and tables for calculating the waiting time, turnaround time, and response time as well as the averages of all results have also been presented. From the data extracted from the presented example, it can be concluded that SJF is the better of the three scheduling algorithms in this instance.

It is important to note the phrase ‘in this instance’, as different OS require different algorithms better suited to the required needs. For example, from the theory presented, it is evident RR is better suited for multitasking systems wanting to utilize time sharing, whereas SJF would be more appropriate for batch OS.

Therefore, from the evidence presented, it can be concluded that a good short term scheduling algorithm should possess certain attributes. These being; maximum utilization of the CPU and maximum throughput, with minimum context switching, minimum waiting time, as well as minimum turnaround time.

2. Specification of System

2.1 Author
Caner Sakallilar.

2.2 Reviewer
The stakeholder to review this design specification is Dr George Charalambous.

2.3 Functionality of System
The system will be a Bash script that emulates the behavior of a round robin scheduling algorithm. The script will read data from a given input that the user specifies as a positional argument in the form of a data file.

2.4 User Interface
It is assumed the user will access and execute the system via their shell terminal in a UNIX based system (see 'Appendix C' for further explanation of UNIX). The user will also provide a data file as mentioned above in '*2.3 Functionality of System*', with the data file containing:

1. A label for each process within the file
2. A burst time value for each process (the NUT value)
3. An arrival time for each process within the file.

The script will then read this data provided from the user and will execute and emulate a round robin algorithm, producing an output on the terminal similar to seen in '*Table 9: RR output' - see 'Appendix B) 8.2iii RR'*. The emulation will display the behavior of the algorithm showing the condition of each process at each time interval:

- '-' a process not yet in the queue.
- 'R' is a process currently in 'running state'.
- 'W' is a process currently in 'waiting state'.
- 'F' is a process that has completed executing, therefore in 'finished state'.

Another such example is showcased in '*Table 10*' below.

Time	P1	P2	P3
0	-	R	W
1	-	W	R
2	W	R	W
3	R	F	W
4	F	F	R
5	F	F	F

Table 10: Proposed output from client (example)

Data for table reference: (Charalambous, G. AS, 2022)

2.5 Goals

Key attributes the system will try to achieve are as follows:

1. Read and store data from file:
The system should test for a set amount of arguments (parameters), ensuring that the filename input from the user is a regular file. The system should then read the data provided from the input file and store this within an array.
2. Implement the algorithm for a quanta value of 1:
The system should loop over time whilst adding and setting processes where arrival time is equal to the real time to the end of the list already existing (from the first element in the array therefore index 0, and set the status of the process to 'W' - waiting state). The process first at the top should be set to 'R' - running state, and decrement burst time (NUT). if the top process has NUT equal to 0 then the status of that process should be set to 'F' - finished state. The status of each process should be printed out and displayed in order of the headers. Once all processes have been tested to check whether they have completed executing then the system should exit from the loop. If some processes are yet to be completed then the top process should be placed in the back of the existing queue (either by loop or setting an index value) and stop this process when the burst time (NUT) is greater than 0.
3. Output correct sequence to standard output and to a file:
The system should output the same order of headers in which the user has input, whilst ensuring that the correct symbolism is displayed. This being: - (not yet in the queue), R (running state), W (waiting state), and F (finished state). The time should also be displayed with the correct process states being displayed under the correct header. For a visual representation of this see *'Table 9: RR output'* and *'Table 10: Proposed output from client (example)'*. The system should also output to standard output and a named file
4. Option for different quanta values:
If viable, the system should take the third parameter input from the user and set such input as the new quanta level after validating. The system should sit the process at the top of the queue to set the new quanta level (from the already set quanta level of one) and move the process if it is completed before the new validating quanta level expires.

Table 11: Specification of System (key attributes)

Data for table reference: (Charalambous, G. AS, 2022)

2.6 Timeline

Date issued: 08/11/2021
Date due: 01/12/2022

2.7 Constraints

The main issue stemming from the specification of the system will be that of a time constraint. To alleviate this, small targets will be set and achieved throughout the time period stated in '2.6 Timeline'. The four

main goals discussed in ‘2.5 Goals’ will be broken down into smaller goals and tested for fluidity of the design and implementation of the system in order to meet the deadline.

2.8 Expectations

The key expectation set out in this specification of the system will be to produce a Bash script that emulates the behavior of a round robin scheduling algorithm by the given deadline.

2.9 Proposed Solution

For a proposed solution of the specification of the system see ‘3. Design of System’.

3. Design of System

3.1 Introduction

The design of the system will highlight an overview of the system itself, showcasing the logical architecture which the script will follow. The system design will also provide a description of key variables utilized (with the requirements of both pre and post conditions of these variables being discussed). Potential design constraints, along with alternative and viable design components will also be evaluated.

3.1i Purpose

The main purpose from this segment of the report is to aid in the readability and implementation of the Bash script.

3.2 Logical System Architecture (Design)

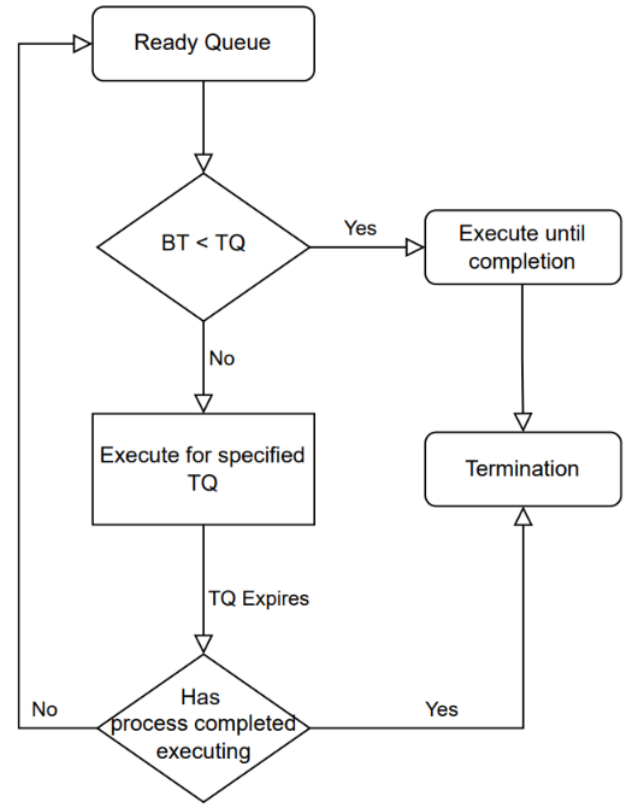


Diagram 1: RR Flowchart

Process not yet in the ready queue	
Pre condition:	(AT != Time) then output '-'
Post condition:	(AT == Time) && (ready queue is empty) then output 'R' (AT == Time) && (ready queue is not empty) then output 'W'
Process in the ready queue	
Pre condition:	(BT !< TQ) then output 'W'
Post condition:	(BT < TQ) && (is first in ready queue) then output 'R' (BT < TQ) && (is ! first in ready queue) then output 'W' (BT == 0) then output 'F'

Table 12: Key Variables - Pre and Post Conditions

The table above (see '*Table 12: Key Variables - Pre and Post Conditions')* highlights a simplified version of the key variables along with their respective outputs pre and post conditions, with the diagram on the left ('*Diagram 1: RR Flowchart')* depicting a simplified version of the logical architecture for the design of the round robin algorithm which is to be implemented. As is displayed, the flowchart begins from a ready queue. For a process to first be placed into the ready queue there must be another array of all processes separate to a ready queue array. Only when a process's arrival time is equal to the real time (AT == Time) is when said process will be placed into the ready queue.

Following on from being placed into the ready queue, for a process in the top position of the ready queue that has a burst time less than that of the value of the time quantum (BT < TQ) then the process will execute - displaying R as the process is now in running state - until its eventual completion and thus terminate (BT == 0). This will remove the process from the ready queue and place it in the finished state (F). This will lead to the next process, if there are still incomplete processes, being shifted to the top of the ready queue, thus changing their state from either '-' or 'W' to now 'R'.

However, for the same process in the top position of the ready queue that does not have a burst time less than that of the value of the time quantum (BT ! < TQ) then it will execute until the time quantum expires and so will be in running state (R). After this instance however, it will be placed back into the ready queue if its NUT - or burst time - is not equal to 0 (BT != 0) and will be in waiting state (W). Respectively, if said process has now completed executing with (BT == 0) then it will be removed from the queue as the process has now terminated and its state will change from running state (R) to finished state (F).

This whole process will loop over time and repeat until all processes are in the finished state (F), thus resulting in, or breaking, a condition to exit the loop and therefore terminating the script.

3.3 Viable Alternatives

As discussed the design will store all elements of the data file within a single array, updating the array where needed. However, separate arrays for each element of the data file with an index of plus three could also be utilized and implemented, this way would promote the use of numerous nested if statements and loops potentially making the script less robust and worsening the readability albeit providing the same output.

3.4 Design Constraints

The main issues stemming from the design of the system will be that the program can not execute if the wrong number of arguments (parameters) are input by the user. If this is the case then a small prompt instructing the user on what has happened and why will alleviate this. Similarly, the file must be a regular file otherwise the program will again not execute.

4. Implementation

For the commented code and video demonstration see *'Appendix D'*.

4.1 Pseudo Code

- ➔ Test for the correct number of parameters
- ➔ if parameters less than one then
  - ◆ Incorrect number of parameters entered
- ➔ Test the file is a regular file
- ➔ if it is not then
  - ◆ echo it does not exist
  - ◆ else echo name of the file back to the user stating it has been entered
- ➔ #Now to clarify the value of the time quant
- ➔ ValueTimeQuant is equal to one
- ➔ if the number of parameters entered is equal to two then
  - ◆ the value of the time quant shal be updated to this by using ValueTimeQuant equals $2
  - ◆ echo what has happened back to the user
- ➔ #example: A 1 2 (A is process number, 1 is arrival time, 2 is burst time (NUT value)
- ➔ declare all arrays making names have integer values and create a variable to use as a way to identify iterating through all arrays as well as creating another to take the remainder as there are three columns within each data file
- ➔ While loop to now read the data from the input file and now sort them into an array
- ➔ for every iteration of a word within the input data file add one for spaces in between
- ➔ if remainder of iteration equals one then
  - ◆ store this in the process id array
- ➔ if remainder of iteration equals two then
  - ◆ store this in the time arrival array
- ➔ else store this in nut value (burst time array)
- ➔ done and save all in to a new array for processes ready to print, printing process id at the top of the output eg '${pArray[@]}"
- ➔ declare a new array to save all elements inside of in the form of a list then test if all processes have completed by iterating through each element while looping over time
- ➔ set process when AT==T to the end of the existing list from index 0 then
- ➔ if its current time is the same as the arrival time then
  - ◆ set the status to 'W' for all process and append it into the newly declared list array
- ➔ initially set this loop to not stop until it is required to break it then loop from the first process through all of them setting the top process to 'R'
- ➔ if the nut value (burst time) is still > 0 there is no need to stop for this iteration however the nut value must be decremented by nut value and the process should be saved as 'R' and not 'F', and append it to the back of the list. Repeat this for the next process
- ➔ If the top process NUT value (burst time) == 0 then
  - ◆ display the correct symbol by setting the process state to F thus printing 'F'
- ➔ Else continue to mark it as R if NUT value is still > 0
- ➔ if all elements in the list array are equal to F then there is no need to loop any further
- ➔ if a process has not been added to the list then
  - ◆ print the output at that time (Arrival time > the real time) as '-'.
- ➔ whilst looping over time, move the process if it has completed

Further information on key built in bash commands used can be found in the table below (see '*Table 13: Key Commands with Description*').

Key commands	Description
declare -a “example”	The "example" variable will be interpreted as an array.
declare -i “example”	The script will interpret "example" occurrences as integers.
IFS (internal field separator)	When interpreting character strings, this variable determines how Bash detects fields or word boundaries.
read -r line	One line is received from standard input, and the characters in the IFS variable are used to break the line into words. -r: If this option is selected, the backslash character does not function as an escape character. The backslash is considered a line character. Specifically, a backslash-newline pair cannot be used to continue a line.
expr	Evaluate expressions and report the result to the standard output.
let	The let builtin allows shell variables to undergo arithmetic operations.
printf	Print and format data. Under the control of the format, write the prepared arguments to standard output.
break	The break statement exits the current loop before its customary conclusion. This is done when the number of times the loop must execute is unknown in advance, such is the case when it is dependent upon by user input.
seq 0 1	Print a sequence of integers for the initial to final increment step.

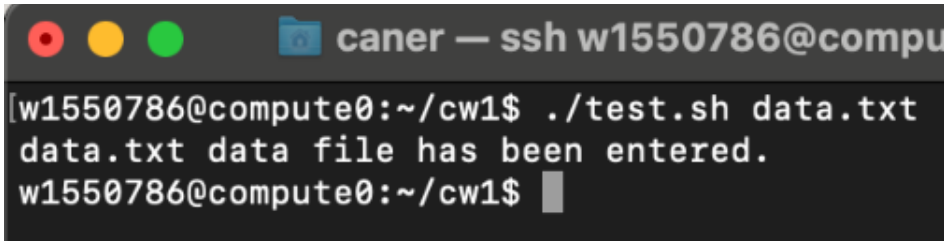
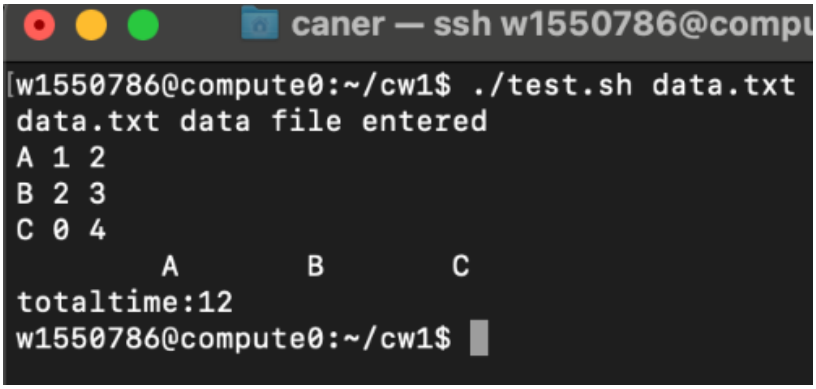
Table 13: Key Commands with Description

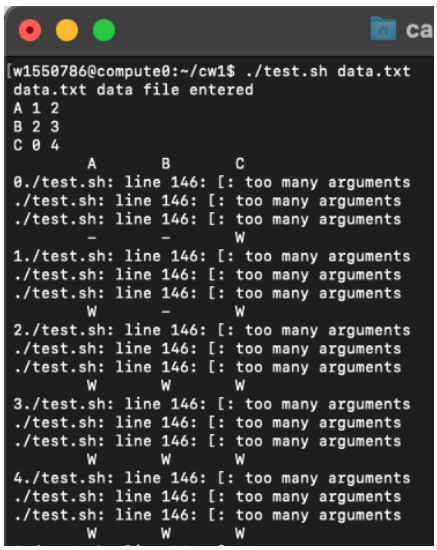
Data for table references: (Linuxtopia, 2022), (ss64, 2022), (TLDP, 2022), (man7, 2022)

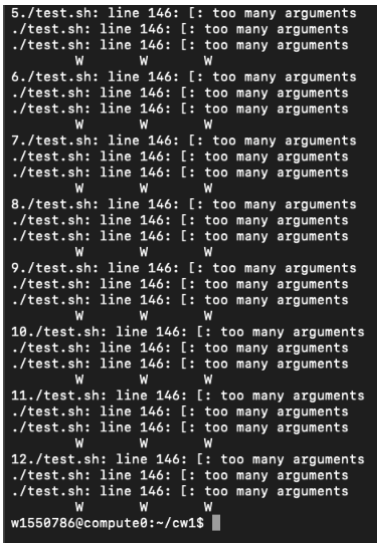
5. Testing

5.1 Results

The table below (‘*Table 14: Testing and Results*’) displays the expected outcomes of certain tests made, with the outcome of each test resulting in a pass or fail. Evidence of a test will be provided with a screenshot below the details of the test and, if a test has failed, retests to rectify the issue will also be provided.

Test	Test Type	Test Input Data	Testing	Expected Outcome	Outcome	Pass/Fail
1	Validation	./test.sh data.txt	For the correct number of parameters and if the file is a regular file.	data.txt data file has been entered.	data.txt data file has been entered.	Pass
						
2	Range	./test.sh data.txt	The previous successful test outcome. Also for this instance, all process labels to be printed out on the screen in order respective to the data.txt along with total time	Same as previous test output with process labels underneath and a total time of 12 being displayed.	data.txt data file has been entered. A 1 2 B 2 3 C 0 4 A B C totaltime:12	Pass
						
3	Range	./test.sh data.txt	The previous successful test outcome with the added process to show each line's respective output at each instance of real time.	For this data file the output should display the whole correct final output along with the number for each step alongside each line of input	An error of too many arguments (as seen in the screenshots below). Also the script seems to be only printing W.	Fail



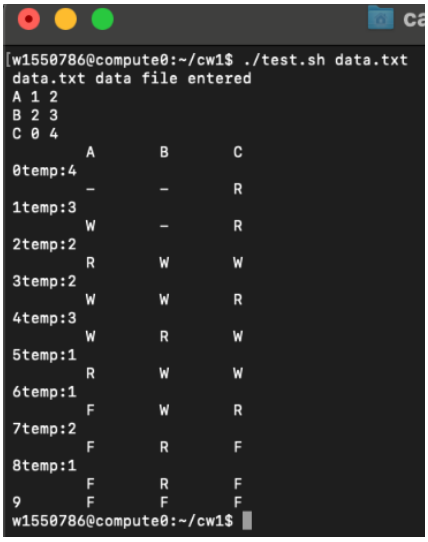


Critical Scenario for test 3 as there is a Point of Failure. A failure here would result in the script not fulfilling the specification by printing the correct output for any of the data files.

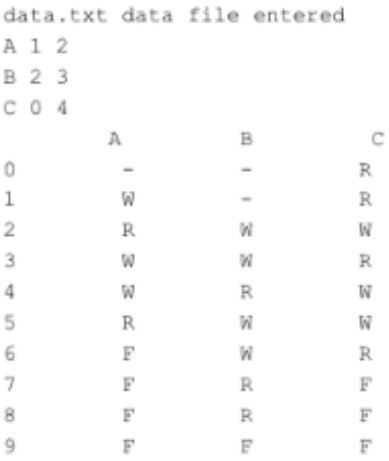
From the failed test it is evident that somewhere in the script it is constantly looping an argument resulting in every process to print ‘W’ and showing all processes are stuck in waiting state. There seems to be no issue with arrival time and also no issue with the NUT value being decremented as the script terminates when it is supposed to, however with the incorrect output. Therefore it is most likely a mistake with the way the script is iterating through the arrays, not allowing the process to change from waiting state.

Upon review of the script, the error was the condition to set the top process to ‘R’ to show that it is in running state.
Error: temp=${tlist[@]}
Solution: temp=${tlist[2]}

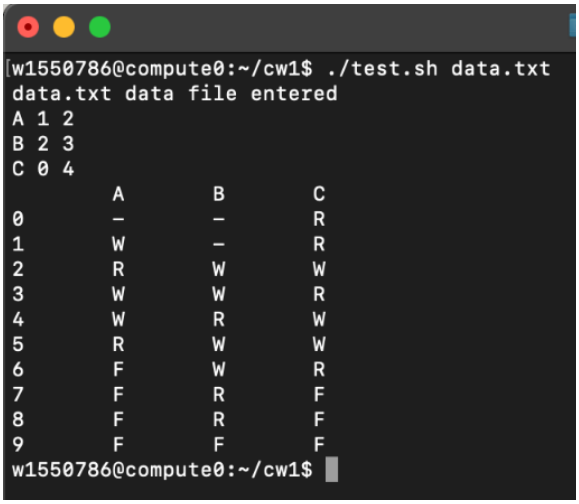
4	Solution for failed test	./test.sh data.txt	For a solution to the point of failure found in ‘Test 3’.	Same expected outcome for ‘Test 3’.	See screenshot below	Pass



5	Output	./test.sh data.txt	Correct output of data file	See screenshot below (left)	See screenshot below (right)	Pass

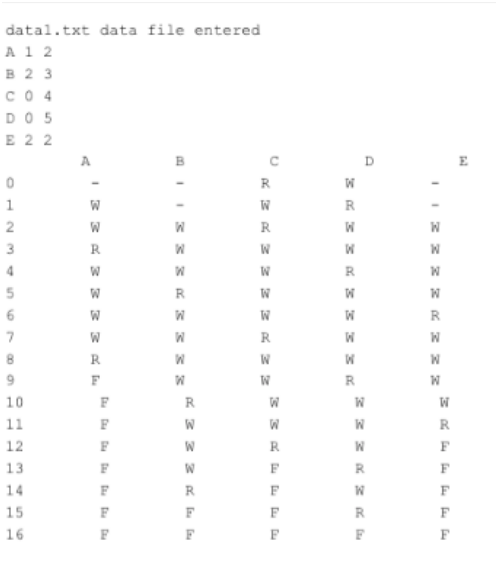


```
data.txt data file entered
A 1 2
B 2 3
C 0 4
      A       B       C
0     -       -       R
1     W       -       R
2     R       W       W
3     W       W       R
4     W       R       W
5     R       W       W
6     F       W       R
7     F       R       F
8     F       R       F
9     F       F       F
```

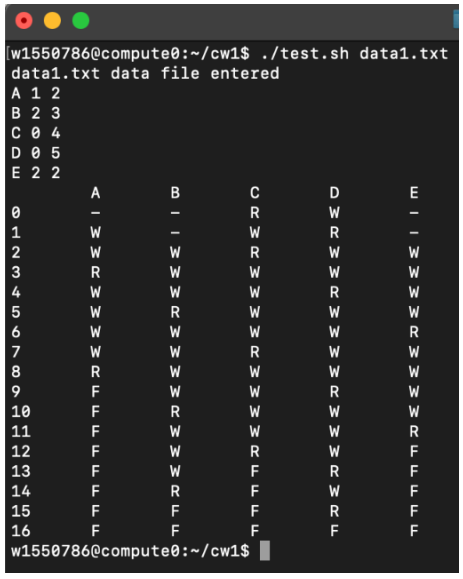


```
w1550786@compute0:~/cw1$ ./test.sh data.txt
data.txt data file entered
A 1 2
B 2 3
C 0 4
      A       B       C
0     -       -       R
1     W       -       R
2     R       W       W
3     W       W       R
4     W       R       W
5     R       W       W
6     F       W       R
7     F       R       F
8     F       R       F
9     F       F       F
w1550786@compute0:~/cw1$
```

6	Output	./test.sh data1.txt	Correct output of data file	See screenshot below (left)	See screenshot below (right)	Pass



```
data1.txt data file entered
A 1 2
B 2 3
C 0 4
D 0 5
E 2 2
      A       B       C       D       E
0     -       -       R       W       -
1     W       -       W       R       -
2     W       W       R       W       W
3     R       W       W       W       W
4     W       W       W       R       W
5     W       R       W       W       W
6     W       W       W       W       R
7     W       W       R       W       W
8     R       W       W       W       W
9     F       W       W       R       W
10    F       R       W       W       W
11    F       W       W       W       R
12    F       W       R       W       F
13    F       W       F       R       F
14    F       R       F       W       F
15    F       F       F       R       F
16    F       F       F       F       F
```



```
w1550786@compute0:~/cw1$ ./test.sh data1.txt
data1.txt data file entered
A 1 2
B 2 3
C 0 4
D 0 5
E 2 2
      A       B       C       D       E
0     -       -       R       W       -
1     W       -       W       R       -
2     W       W       R       W       W
3     R       W       W       W       W
4     W       W       W       R       W
5     W       R       W       W       W
6     W       W       W       W       R
7     W       W       R       W       W
8     R       W       W       W       W
9     F       W       W       R       W
10    F       R       W       W       W
11    F       W       W       W       R
12    F       W       R       W       F
13    F       W       F       R       F
14    F       R       F       W       F
15    F       F       F       R       F
16    F       F       F       F       F
w1550786@compute0:~/cw1$
```

7	Output	./test.sh data2.txt	Correct output of data file	See screenshot below (left)	See screenshot below (right)	Pass

```
data2.txt data file entered
A 1 2
B 3 3
C 1 4
D 2 5
E 3 2
      A     B     C     D     E
0     -     -     -     -     -
1     R     -     W     -     -
2     W     -     R     W     -
3     R     W     W     W     W
4     F     W     W     R     W
5     F     W     R     W     W
6     F     R     W     W     W
7     F     W     W     W     R
8     F     W     W     R     W
9     F     W     R     W     W
10    F     R     W     W     W
11    F     W     W     W     R
12    F     W     W     R     F
13    F     W     R     W     F
14    F     R     F     W     F
15    F     F     F     R     F
16    F     F     F     R     F
17    F     F     F     F     F
```

```
[w1550786@compute0:~/cw1$ ./test.sh data2.txt
data2.txt data file entered
A 1 2
B 3 3
C 1 4
D 2 5
E 3 2
      A     B     C     D     E
0     -     -     -     -     -
1     R     -     W     -     -
2     W     -     R     W     -
3     R     W     W     W     W
4     F     W     W     R     W
5     F     W     R     W     W
6     F     R     W     W     W
7     F     W     W     W     R
8     F     W     W     R     W
9     F     W     R     W     W
10    F     R     W     W     W
11    F     W     W     W     R
12    F     W     W     R     F
13    F     W     R     W     F
14    F     R     F     W     F
15    F     F     F     R     F
16    F     F     F     R     F
17    F     F     F     F     F
w1550786@compute0:~/cw1$
```

8	Output	./test.sh data3.txt	Correct output of data file	See screenshot below (left)	See screenshot below (right)	Pass

```
data3.txt data file entered
A 1 2
B 3 3
C 3 1
D 4 2
E 3 2
      A     B     C     D     E
0     -     -     -     -     -
1     R     -     -     -     -
2     R     -     -     -     -
3     F     R     W     -     W
4     F     W     R     W     W
5     F     W     F     W     R
6     F     R     F     W     W
7     F     W     F     R     W
8     F     W     F     W     R
9     F     R     F     W     F
10    F     F     F     R     F
11    F     F     F     F     F
```

```
[w1550786@compute0:~/cw1$ ./test.sh data3.txt
data3.txt data file entered
A 1 2
B 3 3
C 3 1
D 4 2
E 3 2
      A     B     C     D     E
0     -     -     -     -     -
1     R     -     -     -     -
2     R     -     -     -     -
3     F     R     W     -     W
4     F     W     R     W     W
5     F     W     F     W     R
6     F     R     F     W     W
7     F     W     F     R     W
8     F     W     F     W     R
9     F     R     F     W     F
10    F     F     F     R     F
11    F     F     F     F     F
w1550786@compute0:~/cw1$
```

9	Output	./test.sh data4.txt	Correct output of data file	See screenshot below (left)	See screenshot below (right)	Pass

```
data4.txt data file entered
A 1 2
B 3 3
C 3 1
D 9 4
E 10 2
        A       B       C       D       E
0       -       -       -       -       -
1       R       -       -       -       -
2       R       -       -       -       -
3       F       R       W       -       -
4       F       W       R       -       -
5       F       R       F       -       -
6       F       R       F       -       -
7       F       F       F       -       -
8       F       F       F       -       -
9       F       F       F       R       -
10      F       F       F       R       W
11      F       F       F       W       R
12      F       F       F       R       W
13      F       F       F       W       R
14      F       F       F       R       F
15      F       F       F       F       F
```

```
w1550786@compute0:~/cw1$ ./test.sh data4.txt
data4.txt data file entered
A 1 2
B 3 3
C 3 1
D 9 4
E 10 2
        A       B       C       D       E
0       -       -       -       -       -
1       R       -       -       -       -
2       R       -       -       -       -
3       F       R       W       -       -
4       F       W       R       -       -
5       F       R       F       -       -
6       F       R       F       -       -
7       F       F       F       -       -
8       F       F       F       -       -
9       F       F       F       R       -
10      F       F       F       R       W
11      F       F       F       W       R
12      F       F       F       R       W
13      F       F       F       W       R
14      F       F       F       R       F
15      F       F       F       F       F
w1550786@compute0:~/cw1$
```

10	Validation	./test.sh data.txt 2	For the third parameter to be read and validated as the new NUT value (burst time)	The time quant value has now been set to:2	The time quant value has now been set to:2	Pass

```
w1550786@compute0:~/cw1$ ./test.sh data.txt 2
data.txt data file entered
The time quant value has now been set to:2
A 1 2
B 2 3
C 0 4
```

11	Output	./test.sh data.txt 2	Correct output of data file with newly validated NUT value (burst time)	Same as the previous test output with the final output of the data file correctly displayed	See screenshot below	Fail

```
w1550786@compute0:~/cw1$ ./test.sh data.txt 2
data.txt data file entered
The time quant value has now been set to:2
A 1 2
B 2 3
C 0 4
        A       B       C
0       -       -       R
2       -       W       R
4       -       R       W
6       -       W       R
8       -       R       W
10      -       W       R
12      -       R       F
w1550786@compute0:~/cw1$
```

Critical Scenario for test 11 as there is a Point of Failure. A failure here would result in the script not fulfilling the final point of the specification (albeit this final point being a way for additional marks to be obtained). Half the process is complete, with Test 10 having a Pass outcome, however the second half of this process has not been fulfilled. I believe based on the outcome the issue is with how the script is iterating through all elements. Due to time constraints, it would be more viable to present the product to the client without this aspect. Constraints on this particular point will be discussed further in the evaluation of the built system below (see '6. Evaluation of Built System').

Table 14: Testing and Results

Data for expected outcome reference: (Charalambous, G. TDO, 2022)

6. Evaluation of Built System

From the previous section of this report (5. Testing) it is evidently clear that the built system achieves the key expectation set out in the specification of the system (‘2.8 Expectations’), which is the production of a Bash script that emulates the behavior of a RR scheduling algorithm. The system successfully achieves three out of the four the key goals, just falling short on completion of the fourth goal - albeit not affecting the functionality of the script unless a time quantum other than one is set, as depicted in the table below (see ‘*Table 15: Goals Achieved’).*

Goal	Achieved
Read and store data from file	Yes
Implement the algorithm for a quanta value of 1	Yes
Output correct sequence to standard output and to a file	Yes
Option for different quanta values	No

Table 15: Goals Achieved

The system does read and validate the third parameter input from the user however when the time quantum is manually changed in this manner from the already pre-set condition of one, the output of the script is not correct. This will only play a part in the robustness of the code if the user wants to input a time quantum value other than one. The time constraint already highlighted in ‘2.7 Constraints’ played a factor into achieving this final goal, as with more time this would most likely have been achieved. This being said however, the product follows the specification of the system as well as the implementation of the design of the system to a high degree and does achieve all other requirements set out by the user. This statement is supported by both the detailed qualitative evidence, as well as the quantitative evidence presented throughout this report.

7. Bibliography

I. Behzad, S., Fotohi, R. and Effatparvar, M., 2013. Queue based job scheduling algorithm for cloud computing. *International Research Journal of Applied and Basic Sciences ISSN*, *37853790*.
In text: (Behzad, S. Et al, 2013)

II. Adekunle, Y.A., Ogunwobi, Z.O., Jerry, A.S., Efuwape, B.T., Ebiesuwa, S. and Ainam, J.P., 2014. A comparative study of scheduling algorithms for multiprogramming in real-time systems. *International Journal of Innovation and Scientific Research*, *12*(1), pp.180-185.
In text: (Adekunle, Y.A. Et al, 2014)

III. Charalambous, G. (2022) 'Assignment Specification' [Assessment-Coursework] *7SENG012W Assignment Specification 2022/23.* University of Westminster. 8 November.
In text: (Charalambous, G. AS, 2022)

IV. *On-line Linux and Open Source Technology Books and How To Guides* (Accessed: 2022). Available at: https://www.linuxtopia.org/.
In text: (Linuxtopia, 2022)

V. *SS64 Command line reference* (Accessed: 2022). Available at: https://ss64.com/.
In text: (ss64, 2022)

VI. *The Linux Documentation Project* (Accessed: 2022). Available at: https://tldp.org/.
In text: (TLDP, 2022)

VII. *Michael Kerrisk - man7.org* (Access: 2022). Available at: https://man7.org/.
In text: (man7, 2022)

VIII. Charalambous, G. (2022) 'Test data with output' [Assessment-Coursework] *7SENG012W 2022/23.* University of Westminster. 8 November.
In text: (Charalambous, G. TDO, 2022)

8. Appendices

8.1 Appendix A) *Criteria defined*

1. **Utilization of CPU**: The average amount of time the processor is engaged.

2. **Burst Time**: Burst time is the total amount of time a process needs in order to finish executing on the CPU.

3. **Arrival Time**: The moment a process enters the ready queue is referred to as its arrival time.

4. **Completion Time**: The moment a process enters the completion state or completes executing is referred to as completion time.

5. **Waiting Time**: The total amount of time a process spends in the waiting state in the ready queue.

6. **Response Time**: Response time is the amount of time it takes to allocate a work to the CPU for the first time from the moment it enters a new state. Minimum response time improves algorithm performance.

7. **Throughput**: Throughput refers to the quantity of work performed in a given length of time. More work is performed by the system the bigger the quantity of completed jobs. A more efficient method must have a higher throughput.

8. **Turnaround Time**: The duration between a process's time of arrival and completion is referred to as its turnaround time. Considered superior is an algorithm with a shorter response time.

9. **Fairness**: Prevent famine from occurring. There must be equal opportunity for the execution of all processes involved..

8.2 **Appendix B)** *Quantitative results (with Gantt charts)*

A Gantt chart is a type of bar chart that gives a visual representation of the time-scheduled project tasks. It represents the project's timeline, which may contain the order of tasks to be completed, the total time allotted to complete them, and the dates/times on which they are to begin and conclude, is depicted by a horizontal bar chart with varying lengths.

The Gantt chart indicates tasks that may be performed concurrently and those that cannot be initiated or completed until other processes have been completed. It can assist in identifying possible bottlenecks and processes that may have been omitted from the timeline of the whole project.

8.2i FCFS

Process	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turnaround Time (TaT)	Waiting Time (WT)	Response Time (RT)
P1	3	1	12	9	8	8
P2	0	1	1	1	0	0
P3	2	5	7	5	0	0
P4	4	2	14	10	8	8
P5	2	4	11	9	5	5

Table 2: FCFS result

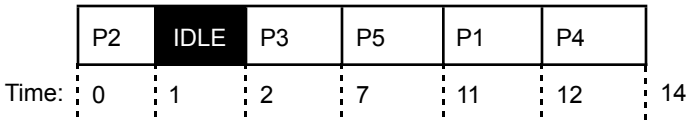


Table 3: FCFS Gantt Chart

8.2ii SJF

Process	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turnaround Time (TaT)	Waiting Time (WT)	Response Time (RT)
P1	3	1	7	4	3	3
P2	0	1	1	1	0	0
P3	2	5	14	12	7	7
P4	4	2	9	5	3	3
P5	2	4	6	4	0	0

Table 4: SJF result

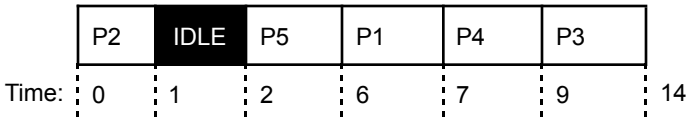


Table 5: SJF Gantt Chart

8.2iii RR

Process	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turnaround Time (TaT)	Waiting Time (WT)	Response Time (RT)
P1	3	1	6	3	2	2
P2	0	1	1	1	0	0
P3	2	5	14	12	7	0
P4	4	2	11	7	5	3
P5	2	4	15	13	9	1

Table 6: RR result

	P2	IDLE	P3	P5	P3	P1	P5	P4	P3	P5	P4	P3	P5	P3	P5	
Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Table 7: RR Gantt Chart (See 'Table 8: RR ready queue')

	P2	IDLE	P3, P5	P5, P3, P1	P3, P1, P5, P4	P1, P5, P4, P3	P5, P4, P3	P4, P3, P5	P3, P5, P4
Time:	0	1	2	3	4	5	6	7	8

P5, P4, P3	P4, P3, P5	P3, P5	P5, P3	P3, P5	P5	
9	10	11	12	13	14	15

Table 8: RR ready queue (See 'Table 9: RR output' on the following page)

Time	P1	P2	P3	P4	P5
0	-	R	-	-	-
1	-	F	-	-	-
2	-	F	R	-	W
3	W	F	W	-	R
4	W	F	R	W	W
5	R	F	W	W	W
6	F	F	W	W	R
7	F	F	W	R	W
8	F	F	R	W	W
9	F	F	W	W	R
10	F	F	W	R	W
11	F	F	R	F	W
12	F	F	W	F	R
13	F	F	R	F	W
14	F	F	F	F	R
15	F	F	F	F	F

Time Quanta Value = 1

Table 9: RR output

The above table shows the output of the RR algorithm for this case. It uses the information presented in '*Table 7: RR Gantt Chart*' and '*Table 8: RR ready queue*' (extension of '*Table 7: RR Gantt Chart*') to show a visualization of the processes and their states at each time instance;

- '-' being not in the queue
- 'R' being in running state
- 'W' being in waiting state
- 'F' being in finished state

8.3 **Appendix C)** *Further explanation of UNIX*

UNIX is an operating system which has undergone continuous development ever since its creation in the 1960s. Operating system, or OS, refers to the collection of programmes that allow a computer to function. It is a multi-user, multitasking, and robust system.

The operating system of UNIX consists of three key components:

1. The **kernel**:
   The UNIX kernel is the operating system's central processing unit; it distributes time and memory to applications and manages the filesystem and communications in response to system calls. As an example of the interaction between the shell and the kernel, imagine a user inputs rm datafile (this command will remove the file called datafile). Using system calls, the shell scans the filestore for the file which contains the program rm and then asks the kernel to perform the program rm on datafile. When the rm datafile process completes, a UNIX prompt returned via the shell back to the user is issued to show the user that the shell is awaiting further instructions (inputs).

2. The **shell**:
   The shell functions as an interface between the user and the kernel. When a user signs in, the login software verifies the login credentials, and then launches the shell. Shell is a CLI, otherwise known as a command line interpreter. It evaluates the instructions input by the user and organizes for their execution. The commands are programs in and of themselves; when they finish executing and so come to an end, the user is presented with another prompt from the shell.

3. The **programs (file system)**:
   It is key to note that everything in UNIX is a file or process. An active programme that is distinguished by its own PID is referred to as a process (process identifier). A collection of information is known as a file. Users construct them by utilizing various tools, such as text editors, compilers, and so forth.

8.4 **Appendix D)** *Commented code (with video demonstration link)*

Video demonstration link:
https://drive.google.com/file/d/1ODsricZxIFtruDKUuWTHQ1BCfu__-kvt/view?usp=share_link

Commented code:

```
#!/bin/bash
ParameterCount=$#

#testing for correct number of parameters
if [ $ParameterCount -lt 1 ]; then
   echo "Incorrect number of parameters entered."
   exit
fi

NameDataFile=$1
#test that NameDataFile is regular
if ! [ -f "$NameDataFile" ]; then
   echo "$NameDataFile does not exist."
   exit
else
   echo "$NameDataFile data file entered"
fi

ValueTimeQuant=1
#read third parameter and validate
if [ $ParameterCount -eq 2 ];then
        ValueTimeQuant=$2
        echo "The time quant value has now been set to:$ValueTimeQuant"
fi

#A 1 2
#A is process number
#1 is arrival time
#2 is NUT value

#declare -A lineArray
declare -a pArray=()      #used for process id
declare -i tArray#used for time arrival
declare -i nArray         #used for nut value
declare -i car=()#used for saving process for printing column

itr=0             #used as identifier while iterating through list/arrays
rem=3             #used for taking remainder; as there are three columns in data.txt. process id, arrival
time, nut value
numProcess=0  #initially total process as zero
column=0        #initially total column as zero

#read and sort in array
```

```
while IFS= read -r line
do
	echo "$line"
	let "numProcess=numProcess+1"
	for word in $line
	do
		let "itr=itr+1"
	# if its first column read as process
	if [ `expr $itr % $rem` -eq 1 ];
	then
		pArray+=($word)
	#			pArray=(${pArray[@]} $word)
	elif [ `expr $itr % $rem` -eq 2 ];
	then
		#read time arrary
		tArray+=($word)
	else
		nArray+=($word)
	fi
	done

  #save column
  car+=($column)

  let "column=column+1"

done < "$NameDataFile"

#printing process ids at top
for i in "${pArray[@]}"
do
	printf "\t%s" "$i"
done

echo ""
totaltime=0	#used for saving nut values+arrival time sum

for i in "${nArray[@]}"
do
	let "totaltime=totaltime+$i"
done

#add total time from time arrival of file
for i in "${tArray[@]}"
do
	let "totaltime=totaltime+$i"
done

#echo "totaltime:$totaltime"
```

```
tt=0

declare -a tlist=()          ###tlist is main array, it is used for saving all the elements in a single array

titr=0    #used as identifier while iterating through list/arrays
np=0    #total process-1
let "np=numProcess-1"
time=0  ###value as time value for looping in the processes
abc=0   #variable as an identification for looping that this process has completed, it should not be marked as F but should be marked as R

colid=0
        #test all processes completed
        while [ $time -le $totaltime ]                          #loop all over the time
        do
                litr=0
                process=0
                #display time
                printf "%s" "$time"

                #add/set process that AT==T to the end of the existing list initially from index 0 & set status to W
                for i in "${tArray[@]}"
                do
                        if [ $i -eq $time ]; then    #if its the current time; which is same as arrival time of the process
                                process=${pArray[$litr]} #get process
                                time=$i                  #
                                nut=${nArray[$litr]}             #get nut value
                                clmn=${car[$litr]}               #get column
                                tlist+=("$process" "$time" "$nut" "W" "$clmn" "$abc" )      #append it in tlist
                                ((colid++))
                        fi
                        ((litr++))
                done

                itrr2=0
                for pr2 in "${tlist[@]}"
                do
                        if [ `expr $itrr2 % 6` -eq 3 ];        ##get 4 thr column of tlist
                        then
                                tlist[$itrr2]="W"  ##save all of the process as W, at startup
                        fi
                        let "itrr2=itrr2+1"                     #increment iterator value
                done

                dobreak=0       #make the loop to stop when required, initially set it to don't stop
```

```
		for n in $(seq 0 1 $np)	#loop from 1st process to all
		do
			itr=0
			for pr in "${tlist[@]}"
			do
				#set top process to R
				if [ $itr -eq 0 ];then
					temp=${tlist[2]}
					if [ $temp -gt 0 ];then		#if nut value value is greater than 0; it means there is need to stop for this iteration
						#echo "temp:$temp"
						tlist[3]="R"			#mark it as read
						#decrement NUT
						let "temp=temp-1"		#decrement nut
						tlist[2]=$temp
						abc=1				#variable for saving it as R; don't change it to F
						if [ $temp -eq 0 ];then
							tlist[5]=$abc
						fi
						dobreak=1		#nut was greater than 0, no need to check next process
					fi

				fi
				let "itr=itr+1"

			done

			#append this to end of list
			itrr=0
			for pr1 in "${tlist[@]}"
			do
				if [ $itrr -eq 0 ];then	#save this and move it to end of array
					p=${tlist[0]}
					t=${tlist[1]}
					n=${tlist[2]}
					r=${tlist[3]}
					c=${tlist[4]}
					a=${tlist[5]}
					tlist+=("$p" "$t" "$n" "$r" "$c" "$a")	#appended to end
				fi
				let "itrr=itrr+1"
			done
			#remove this from start of list
			#move the index
			tlist=("${tlist[@]:6}")	#removed from start
```

```
                        #update column
                        itrr2=0

                        for pr2 in "${tlist[@]}"
                        do
                                if [ `expr $itrr2 % 6` -eq 2 ];         #if its 3rd column of list
                                then
                                        if [ $pr2 == 0 ];then           #if nut  is zero;;; used for marking it F or
R based on abc variable marked above
                                                dit=0
                                                let "dit=itrr2+1"
                                                acheck=0
                                                let "acheck=itrr2+3"
                                                ac=${tlist[$acheck]}
                                                if [ $ac -eq 0 ];then
                                                        #test top process to F if nut ==0
                                                                                        #display correct
symobolism
                                                        tlist[$dit]="F"
                                                else
                                                        #display correct symobolism
                                                        tlist[$dit]="R"
                                                        ac=0
                                                        tlist[$acheck]=$ac          #read this as R, but in
future read it as F
                                                fi
                                        fi

                                fi
                                let "itrr2=itrr2+1"
                        done

                        if [ $dobreak -eq 1 ];then
                                break;              #if above dobreak was set as 1, then stop this loop
                        fi

                done

                fitr=0

                for n in $(seq 0 1 $np)    #iterate over all the processes
                do
                        citr=0
                        c0=0

                        #correct header output based on header input
                        for pr in "${tlist[@]}"
                        do
                                if [ `expr $citr % 6` -eq 4 ];then    #read 5th column of tlist
```

```
                                        acs=$(expr $pr % $numProcess)
                                        #echo "pr:$pr np:$numProcess acs::$acs"
                                        #print out process in order of header
                                        if [ $acs == $n ];then                  #if this the process which was
read as respective process from file. print it under that
                                                abc=0
                                                let "abc=citr-1"
                                                #output stdout
                                                printf "\t%s" "${tlist[$abc]}"
                                                printf "\t%s" "${tlist[$abc]}" >> NAMEDFILE
                                                c0=1

                                                if [ ${tlist[$abc]} == "F" ];then
                                                        let "fitr=fitr+1"
                                                fi
                                        fi
                                fi
                                let "citr=citr+1"
                        done

                        if [ $c0 -eq 0 ];then
                                #display correct symobolism
                                printf "\t%s" "-"   #if this process is not added to list, it means that arrival
time of this process has not reached, print as -
                                                                        #output stdout
                                printf "\t%s" "-" >> NAMEDFILE
                        fi
                done

                        echo ""

                        if [ $fitr -eq $numProcess ];then  #all the process has F, no need to further loop
                                exit
                        fi
        #loop over time
        let "time=time+ValueTimeQuant"#move process if it completes

        done
```