TITLE: SELFISH ROUND ROBIN SCHEDULING ALGORITHM.

NAME: JOSHUA ADDAI-MARNU

STUDENT NUMBER: 19548571

COURSE: SOFTWARE DEVELOPMENT ENVIRONMENT

DATE: 13/12/2023

TABLE OF CONTENT

INTRODUCTION (UNDERSTANDING OF SRR ALGORITHM) …… 2

IMPLEMENTATION OF SRR …… 3-4

REQUIREMENTS OF THE ALGORITHM …… 5-6

DESIGN OF THE SYSTEM …… 7-15

TESTING ……16-18

CRITICAL EVALUATION ……19-20

BASH SCRIPT WITH COMMENTS ……21-29

LINK TO VIDEO OF TESTING THE SRR ALGORITHM SCRIPT AND REFERENCES …… 30

INTRODUCTION

Scheduling algorithms in operating systems establish the sequence in which processes are executed, taking into account parameters like arrival time and priority. These algorithms oversee the progression of processes from the waiting state to the ready queue, utilizing preemptive or non-preemptive strategies. Preemptive algorithms allow higher-priority processes access to the CPU, interrupting lower-priority processes already in execution. In contrast, non-preemptive scheduling does not interrupt processes once they begin execution, even if higher-priority processes are ready.

The traditional round-robin scheduling algorithm is preemptive, assigning each process a fixed time quantum for execution. When a process completes its time slice, it relocates to the end of the queue, and subsequent processes in the ready queue adhere to a First-Come-First-Serve (FCFS) sequence.

The Selfish Round-Robin (SRR) algorithm, an improved version of the traditional approach, enhances its functionality. It prioritizes processes with prolonged or extended execution times and high priority, ensuring they obtain or receive the required Central Processing Unit(CPU) time. The selfish round-robin algorithm is designed to optimize the implementation beyond the standard round-robin method.

SRR aims to provide better service to ongoing processes compared to new ones. In SRR, processes in the ready list are divided into two groups, that is NEW and ACCEPTED. New processes wait, while accepted processes are served using Round Robin. The priority of a new process increases at a value of “A” and the priority of an accepted process increases at a value of “B”.

IMPLEMENTATION

- Processes in the ready list (when a process arrival time is up) will be grouped into two, that is, new and accepted queues.

- If there are no processes in the accepted queue, then the new process or processes are moved or allocated directly to the accepted queue.

- If there are processes already in the accepted queue, then the new process will wait in the new queue while the accepted processes are serviced (by round robin).

- All new processes are allocated a priority value of “A”, while the priority of the accepted processes is increased by a value of “B”.

- All accepted processes in this system have the same priority, and this priority increases over time at a certain value “B”. This ensures that every new process will eventually get accepted, preventing any process from being ignored for too long.

- At the end of a time cycle, there is an increase in the priority value of the processes in the new and accepted queue by values of “A” and “B” respectively.

- When the priority of a new process reaches the priority of an accepted process, that new process becomes accepted and it is moved to the bottom of the accepted queue.

- Once a process at the top of the accepted queue runs, there is a decrease in service time (NUT) by 1.

- If the process that has just run its service time (NUT) is not equal to zero, then it is moved back to the bottom of the accepted queue.

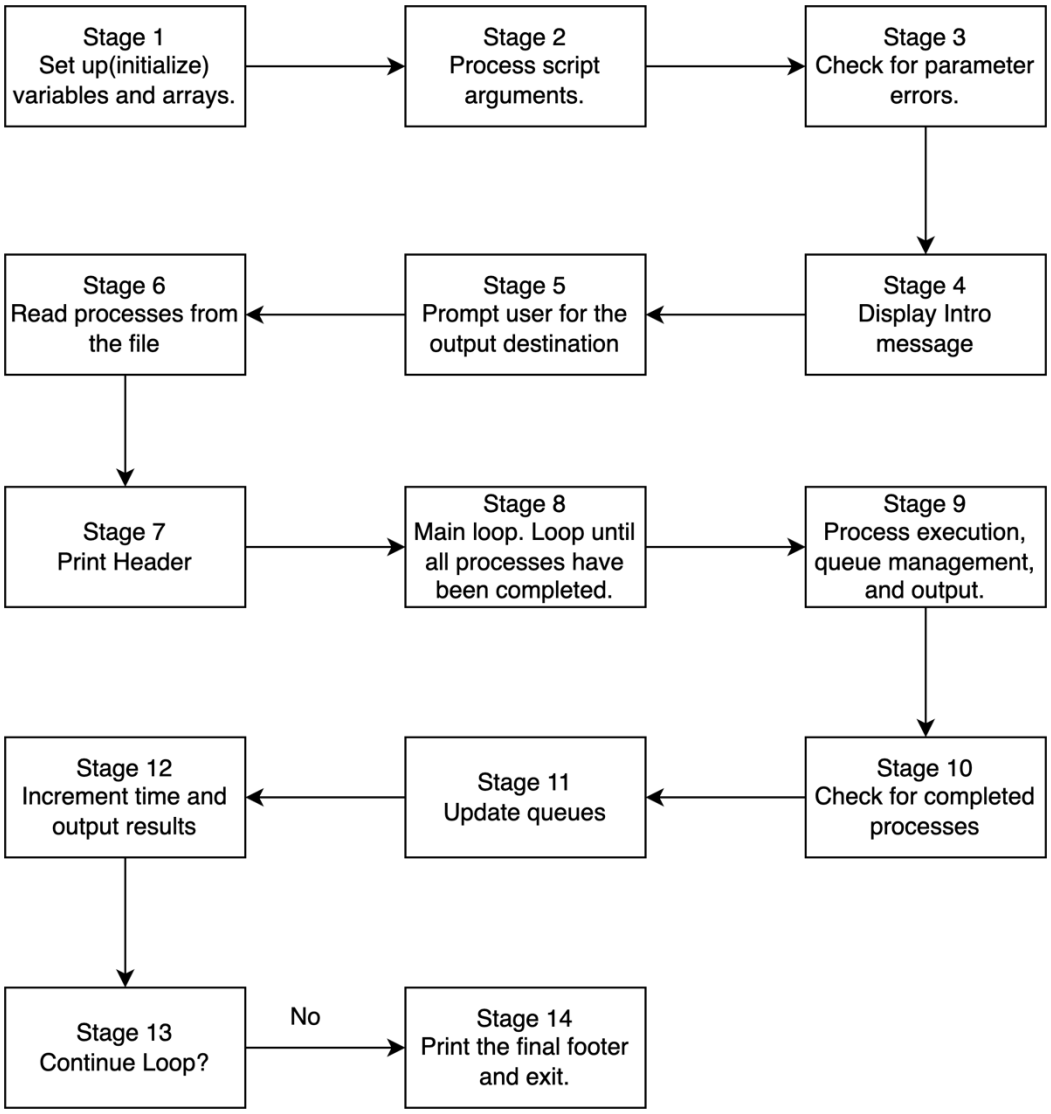
- Once a process in the accepted queue runs out its service time (NUT) to zero, it is deemed as finished and then removed from the accepted queue.

REQUIREMENT OF THE ALGORITHM

REQUIREMENTS	TYPES	REQUIRES/MANDATORY	DESCRIPTION
Positional Parameters	-Data file name (first parameter). -increment value of the new queue (second parameter). -Increment value of the accepted queue (third parameter). -An optional fourth positional parameter for the quanta number.	It is expected that the bash script has the capability to receive three required parameters and an optional fourth parameter.	The data file needs to be created in a specific format. This file should be a regular text file that includes the process names, the exact time it arrived(arrival time), and the amount of time it requires to be serviced(service time). The incremental value of the new queue must be an integer and has to be greater than or equal to two. The incremental value of the accepted queue must be an integer and must be less than the incremental value of the new queue.
User Prompt	-Standard output only -Named text file (overwrite if it exists) -Both standard output and a named text file.	The script should require the user to choose between the output options	The user has to indicate where to submit the output of the results from the available options, that is standard output only(stdout), named text file only (file)(overwrite if it exists) and or both standard output and a named text file.
Data File Format	-Process name or label. -NUT value. -Arrival time.	The named data file provided by the user should contain data for each process	In order for the system to function properly, it is necessary to have a text file that contains specific data. This file should include the name of the

			process, the process service(NUT) in this case, and the arrival time of each process. It is essential that this file exists and contains accurate information. If the file is missing or contains errors, it will cause issues with the system's performance and must be addressed immediately.
Emulation Output	-Process running (R). -Process waiting (W). -Process completed (F). -Process not arrived (-).	This is required to showcase the behaviour of SSR algorithm	The script must emulate the behaviour of the selfish round-robin scheduling algorithm and display for each time interval (set quanta to 1 as default if not specified) of the state of each process. The states or stages of the process are Process running as “R”, Process waiting as “W”, Process completed as “F” and Process not arrived as “—".
Termination		Terminate the script is required.	Terminate or stop the script once all the processes have completed or finished.

DESIGN OF THE SYSTEM



The diagram depicts a simplified version of the script's flow. Each box represents a decision point or step, and arrows indicate control flow. The script executes the following steps primarily:

- Initialization: To start, the script arguments will be parsed, initial values will be set, and arrays will be declared.

- **Parameter Error Checking:** Verify the correctness of script parameters, such as the existence of the input file.
- **Main Algorithm**: Iterate through the main algorithm loop until all processes are completed.
- **Process Executions and Queue Management**: Manage processes of the accepted and new queues, update priorities and execute processes.
- **Output Results**: Output the current status of processes at each time unit.
- **Check for Completed Processes**: Verify if any processes have completed their execution.
- **Update Queues:** remove completed processes from the accepted queue and update the new queue.
- **Increment Time and Output Results:** Increase the time unit and output the updated results.
- **Continue Loop?** Check if all processes are completed. If yes, proceed to the final steps and if no, continue to loop.
- **Print final Footer and Exit:** Output the final status of processes, print the footer and exit the script.

More on Design….

Parameter Handling:

The script is designed to ensure that the user provides an adequate number of positional parameters (at least 3) by performing a parameter-handling process. During this process, the parameters are assigned to variable names that help to improve the code's readability and clarity. By doing so, it becomes easier for the user to understand the code and make necessary modifications if needed.

```
# 1. Storing programming arguments into an array.
incValAcceptedQ=1
incValNewQ=2
quanta=1
directOutput=
outputToFile=
processesCompleted=0
```

```
time=0
paddingSpace=10
timeConsumed=0
progArguments=($*)
lengthOfArguments=$#
# 3. CHECKING FOR PARAMETER ERRORS.

# a. Checking if the data file path exist or can be located
if ! [ -e "${progArguments[0]}" ]
then
    displayErrorMsgAndExit "Input error. The file path
${progArguments[0]} does not exist."
fi

# b. Checking if the right number of argument and argument
type was passed
if [ $lengthOfArguments -lt 3 ]
then
    displayErrorMsgAndExit "Sorry😕, three or four arguments
are required."
elif [ $lengthOfArguments -eq 4 ]
then
    if [[ ${progArguments[3]} =~ ^[0-9]+$ ]]
    then
        quanta=${progArguments[3]}
    else
        displayErrorMsgAndExit "Sorry😕, enter the correct
value (The quanta value has
        to be an integer)."
    fi
fi

# c. çhecking if the values passed for argument 2 and 3 are
integers.
if ! [[ ${progArguments[1]} =~ ^[0-9]+$ ]] || ! [[
${progArguments[2]} =~ ^[0-9]+$ ]]
    then
        displayErrorMsgAndExit "Ooops🤔: Argument 2 and 3
values should be integers"
# checking that increment value of new queue is not greater
that accepted queue
elif [[ ${progArguments[1]} -lt ${progArguments[2]} ]]
    then
        displayErrorMsgAndExit "Ooops🤔: The incremental value
of the of 'Argument 2(New Queue)'
        should be greater than the incremental value of
'argument 3(Accepted Queue)'"
else
    incValNewQ=${progArguments[1]}
```

```
    incValAcceptedQ=${progArguments[2]}
fi
```

Data Structure:

The organization and storage of data is known as data structure.

The selfish round-robin algorithm is a technique used in Bash programming. It relies heavily on an important data structure called an Array. Arrays are like containers that can hold different pieces of information, and they are really useful because they are easy to access and use. Bash programming doesn't support more advanced data structures like Classes or multidimensional arrays, so Arrays are a great option.

In this algorithm, different types of information are stored in separate containers (Arrays), and these containers are connected based on their position in the sequence. This helps the algorithm work more efficiently and get the job done faster.

```
# 2.Using multipe arrays(no multidimensional arrays in Bash)
declare -a nameOfProcesses=()
declare -a serviceTimeProcesses=()
declare -a arrivalTimeProcesses=()

declare -a nameOfNewQ=()
declare -a serviceTimeNewQ=()
declare -a priorityNewQ=()
declare -a statusNewQ=()

declare -a nameOfAcceptedQ=()
declare -a serviceTimeAcceptedQ=()
declare -a priorityAcceptedQ=()
declare -a statusAcceptedQ=()
```

Modularization with Functions:

To make the code more organized and easier to understand, we used Bash functions. These functions encapsulate and hide specific logic. This helps to break down the code into smaller, more manageable pieces and promote readability.

```
# 3. FUNCTIONS

# a. Function: For spacing out text properly.
adjustNumSpaces() {
    local numberOfSpaces="$1"
    local time="$2"

    if [ "${#time}" -ge 2 ]; then
        numberOfSpaces=$((numberOfSpaces - ${#time} + 1))
    fi

    echo "$numberOfSpaces"
}

# b. Function: To verify if a process has consumed the amount
quanta required on the CPU.
verifyQuanta() {
    if [ "${serviceTimeAcceptedQ[0]}" -gt 0 ]; then
        ((timeConsumed += 1))
    else
        timeConsumed=$quanta
    fi
}

# c. Function: Outputing result either to stdout, file or
both.
outputResultDestination() {
    local directOutput="$1"
    local content="$2"
    local outputToFile="$3"
    local addParameter="$4"

    if [[ "$directOutput" == "file" || "$directOutput" ==
"both" ]]; then
        # Make sure the directory structure exist.
        mkdir -p "$(dirname "$outputToFile")"
        # Check if added parameter is specified
        if [ "$addParameter" == "addParameter" ]; then
            echo "$content" >> "$outputToFile"
        else
```

```
            echo "$content" > "$outputToFile"
        fi
    fi

    if [[ "$directOutput" == "stdout" || "$directOutput" ==
"both" ]]; then
        echo "$content"
    fi
}

# d. Function: For formatting the process's state.
createHeader() {
    local name="$1"
    local numberOfSpaces="$2"
    echo -n "$name"

    for ((i=0; i<numberOfSpaces; i++)); do
        echo -n " "
    done
}
# e. Function: For printing error and exiting script.
displayErrorMsgAndExit() {
    local errorMsg="$1"
    echo "Error:  $errorMsg"
    exit 1
}
```

Error Handling:

The program makes sure that the user chooses valid options. If the user makes a mistake and enters an invalid choice, the program shows an error message and stops running. Sometimes, the program shows a warning and gives the user another chance to choose a valid option. There is a specific function called "displayErrorMsgAndExit" that is responsible for showing the error message and stopping the program.

```
# e. Function: For printing error and exiting script.
displayErrorMsgAndExit() {
    local errorMsg="$1"
    echo "Error:  $errorMsg"
    exit 1
```

```
}
```

USER PROMPT

The user will be prompted to select an output option using a simple option. Once they have made a selection, the script will validate their choice to ensure it is a valid option.

```
# prompt the user where the output of the results(of the
algorithm) should be displayed.
while true;
do
    echo -e "Which of the following options do you want the
output of the results to be displayed?  [OPTIONS: STDOUT or
FILE or BOTH]"
    read userFeedback

    # Convert userFeedback to lowercase (for case-insensitive
comparison)
    directOutput=$(echo "$userFeedback" | tr '[:upper:]'
'[:lower:]')

    if [[ "$directOutput" == "stdout" || "$directOutput" ==
"file" || "$directOutput" == "both" ]]; then
        # if a correct option is given, exit the loop.
        if [[ $directOutput == "file" || $directOutput ==
"both"  ]]
        then
            while true;
            do
                echo -e "Which filename do you want to store
output of the results?"
                read userFeedback
                if [[ -n "$userFeedback" ]]; then
                    outputToFile="$userFeedback"
                    break
                else
                    echo "CAUTION!: Kindly enter a filename.
Cannot leave this field
                    blank."
                fi
            done
        fi
        break
    else
```

```
        echo "CAUTION!: $userFeedback is not a valid option.
Please provide either
        STDOUT or FILE or BOTH."
    fi
done
```

The program uses a technique called a while loop to make sure that the user enters a correct choice before it moves forward with the next step.

Process reading and Main Algorithm:

The main functionality of the script initiates by reading the processes specified in the data file and segregating them into distinct arrays, namely “nameOfProcesses, serviceTimeProcesse, and arrivalTimeProcesses”. This is achieved by parsing each line into three columns using a space separator.

```
# Retrieve process from the provided path
while read -r line || [[ -n $line ]];
do
  read -r name service arrival <<<"$line"

  # Appending the new process array to the existing process
array
  nameOfProcesses+=("$name")
  serviceTimeProcesses+=("$service")
  arrivalTimeProcesses+=("$arrival")
done < "${progArguments[0]}"
```

Once all the processes have been loaded, a script uses a process called the "selfish round-robin algorithm" to manage them. The script keeps track of time and continuously checks if there are any processes to be performed. If a process has taken up all the time it needs to be completed, it is moved to the end of the line (accepted

queue) and its progress (status and service time) is updated. If a process has not been completed in the specified time(quanta), the script keeps track of how much time is left and moves on to the next process.

The script also keeps track of when new processes are added to the system and assigns them to either the "accepted" queue or the "new" queue. The script also updates the priority of each process and prints the current status of each process.

When a process is completed, the script removes it from the "accepted" queue and updates the count of completed processes. If there are any processes in the "waiting" queue, the script checks their priority against the first process in the "accepted" queue. If a process in the "waiting" queue has a higher priority, it is moved to the "accepted" queue and its progress is updated.

Overall, the script is designed to efficiently manage a large number of processes in a fair and orderly manner.

The script will stop running when all the tasks are done, and it won't show any error messages.

TESTING OF THE SCRIPT.

Quanta = 2
Incremental value of New Queue = 2
Incremental value of Accepted Queue = 1

Process Name	Arrival Time	Service Required	Expected Finish time	Actual Finish Time
Z	2	2	6	6
E	2	4	12	12
L	4	1	10	10
F	6	5	20	20
I	5	3	17	17
S	0	2	2	2
H	1	3	9	9

Hand Calculation of the SRR algorithm.

Time=0

New Queue	Accepted Queue
-	S(R)(1)(1)

Time=1

New Queue	Accepted Queue
H(W)(3)(2)	S(R)(0)(2)

Time=2

New Queue	Accepted Queue
Z(W)(2)(2) E(W)(4)(2)	H(R)(2)(3)

Time=3

New Queue	Accepted Queue
Z(W)(2)(4) E(W)(4)(4)	H(R)(1)(4)

Time=4

New Queue	Accepted Queue
L(W)(1)(2)	Z(R)(1)(5)
	E(W)(4)(5)
	H(W)(1)(5)

Time=5

New Queue	Accepted Queue
L(W)(1)(4)	Z(R)(0)(6)
I(W)(3)(2)	E(W)(4)(6)
	H(R)(1)(6)

Time=6

New Queue	Accepted Queue
L(W)(1)(6)	E(R)(3)(7)
I(W)(3)(4)	H(W)(1)(7)
F(W)(5)(2)	

Time=7

New Queue	Accepted Queue
L(W)(1)(8)	E(R)(2)(8)
I(W)(3)(6)	H(W)(1)(8)
F(W)(5)(4)	

Time=8

New Queue	Accepted Queue
I(W)(3)(8)	H(R)(0)(9)
F(W)(5)(6)	L(W)(1)(9)
	E(W)(2)(9)

Time=9

New Queue	Accepted Queue
I(W)(3)(10)	L(R)(0)(10)
F(W)(5)(8)	E(W)(2)(10)

Time=10

New Queue	Accepted Queue
F(W)(5)(10)	E(R)(1)(11)
	I(W)(3)(11)

Time=11

New Queue	Accepted Queue
F(W)(5)(12)	E(R)(0)(12)
	I(W)(3)(12)

Time=12		Time=13	
New Queue	Accepted Queue	New Queue	Accepted Queue
	I(R)(2)(13)		I(R)(1)(14)
	F(W)(5)(13)		F(W)(5)(14)

Result from the SRR bash script algorithm.

Time	Z	E	L	F	I	S	H
0	-	-	-	-	-	R	-
1	-	-	-	-	-	R	W
2	W	W	-	-	-	F	R
3	W	W	-	-	-	F	R
4	R	W	W	-	-	F	W
5	R	W	W	-	W	F	W
6	F	R	W	W	W	F	W
7	F	R	W	W	W	F	W
8	F	W	W	W	W	F	R
9	F	W	R	W	W	F	F
10	F	R	F	W	W	F	F
11	F	R	F	W	W	F	F
12	F	F	F	W	R	F	F
13	F	F	F	W	R	F	F
14	F	F	F	R	W	F	F
15	F	F	F	R	W	F	F
16	F	F	F	W	R	F	F
17	F	F	F	R	F	F	F
18	F	F	F	R	F	F	F
19	F	F	F	R	F	F	F
20	F	F	F	F	F	F	F

Using the selfish round-robin algorithm (bash program) and calculating manually based on the data set used, the algorithm executes the data set provide above remarkably.

CRITICAL EVALUATION OF SELFISH ROUND-ROBIN ALGORITHM.

ADVANTAGES OF SELFISH ROUND-ROBIN (SRR) ALGORITHM

- **Equality and Fairness**: SRR scheduling is fair to all processes because it gives each process an equal share of CPU time. This makes it a fairer scheduling algorithm and sets it apart from other algorithms that may favour specific processes.

- **Minimal-Overhead**: SRR scheduling is a straightforward algorithm with minimal implementation overhead. It is easy to grasp and put into action, making it well-suited for uncomplicated systems.

- **Effective Usage of CPU:** SRR scheduling helps use the CPU efficiently by letting processes use only the time they require. This lowers the chance of processes getting stuck without a reason, which can boost the overall system's performance.

DISADVANTAGES OF SELFISH ROUND-ROBIN (SRR) ALGORITHM

- **Not Suitable for Real-Time Systems:** SRR scheduling does not work well for real-time systems with strict timing needs. In these systems, it is crucial to make sure important tasks get CPU time promptly, which SRR scheduling cannot guarantee.

- **Ineffective Management of Priority**: SRR scheduling does not handle process priorities properly. It treats all processes the same, which might result in high-priority tasks not getting enough CPU time.

- **Selfishness**: One big issue with SRR scheduling is that it lets processes be selfish and use all their given CPU time, even when

they do not need it. This can cause some processes to hoard CPU time, making the overall system performance go down.

BASH SCRIPT WITH COMMENTS OF THE SELFISH ROUND-ROBIN ALGORITHM.

```
#!/bin/bash
#
# SELFISH ROUND ROBIN ALGORITHM:
#
# It prioritizes processes with prolonged or extended execution timeSlices and high
priority, #
# ensuring they obtain or receive the required Central Processing Unit(CPU)
timeSliceSlice.   #
# The selfish round-robin algorithm is designed to optimize the implementation
beyond         #
# the standard round-robin method.
#
#
#
#
#
# Name: Joshua Addai-Marnu
#
# Date: 13/12/2023
#
# Student_ID: 19548571
#
#
#
################################################################################
############

# 1. Storing programming arguments into an array.
incValAcceptedQ=1   #Incremental value of accepted queue.
incValNewQ=2        #Incremental value of new queue.
quanta=1            # Quanta value is set to one as default.
directOutput=       # Output directly to terminal
outputToFile=       # Output directly to a file
processesCompleted=0
time=0
paddingSpace=10
timeConsumed=0
progArguments=($*)
lengthOfArguments=$#

# 2.Using multipe arrays(no multidimensional arrays in Bash).
declare -a nameOfProcesses=()
declare -a serviceTimeProcesses=()
declare -a arrivalTimeProcesses=()

declare -a nameOfNewQ=()
declare -a serviceTimeNewQ=()
```

```
declare -a priorityNewQ=()
declare -a statusNewQ=()

declare -a nameOfAcceptedQ=()
declare -a serviceTimeAcceptedQ=()
declare -a priorityAcceptedQ=()
declare -a statusAcceptedQ=()

# 3. FUNCTIONS

# a. Function: For spacing out text properly.
adjustNumSpaces() {
    local numberOfSpaces="$1"
    local time="$2"

    if [ "${#time}" -ge 2 ]; then
        numberOfSpaces=$((numberOfSpaces - ${#time} + 1))
    fi

    echo "$numberOfSpaces"
}

# b. Function: To verify if a process has consumed the amount quanta required on
the CPU.
verifyQuanta() {
    if [ "${serviceTimeAcceptedQ[0]}" -gt 0 ]; then
        ((timeConsumed += 1))
    else
        timeConsumed=$quanta
    fi
}

# c. Function: Outputing result either to stdout, file or both.
outputResultDestination() {
    local directOutput="$1"
    local content="$2"
    local outputToFile="$3"
    local addParameter="$4"

    if [[ "$directOutput" == "file" || "$directOutput" == "both" ]]; then
        # Make sure the directory structure exist.
        mkdir -p "$(dirname "$outputToFile")"
        # Check if added parameter is specified
        if [ "$addParameter" == "addParameter" ]; then
            echo "$content" >> "$outputToFile"
        else
            echo "$content" > "$outputToFile"
        fi
    fi

    if [[ "$directOutput" == "stdout" || "$directOutput" == "both" ]]; then
```

```
        echo "$content"
    fi
}

# d. Function: For formatting the process's state.
createHeader() {
    local name="$1"
    local numberOfSpaces="$2"
    echo -n "$name"

    for ((i=0; i<numberOfSpaces; i++)); do
        echo -n " "
    done
}
# e. Function: For printing error and exiting script.
displayErrorMsgAndExit() {
    local errorMsg="$1"
    echo "Error:  $errorMsg"
    exit 1
}

# 4. CHECKING FOR PARAMETER ERRORS.

# a. Checking if the data file path exist or can be located
if ! [ -e "${progArguments[0]}" ]
then
    displayErrorMsgAndExit "Input error. The file path ${progArguments[0]} does not
exist."
fi

# b. Checking if the right number of argument and argument type was passed
if [ $lengthOfArguments -lt 3 ]
then
    displayErrorMsgAndExit "Sorry😕, three or four arguments are required."
elif [ $lengthOfArguments -eq 4 ]
then
    if [[ ${progArguments[3]} =~ ^[0-9]+$ ]]
    then
        quanta=${progArguments[3]}
    else
        displayErrorMsgAndExit "Sorry😕, enter the correct value (The quanta value
has
        to be an integer)."
    fi
fi

# c. çhecking if the values passed for argument 2 and 3 are integers.
if ! [[ ${progArguments[1]} =~ ^[0-9]+$ ]] || ! [[ ${progArguments[2]} =~ ^[0-9]+$
]]
    then
```

```
        displayErrorMsgAndExit "Ooops🤔: Argument 2 and 3 values should be
integers"
# checking that increment value of new queue is not greater that accepted queue
elif [[ ${progArguments[1]} -lt ${progArguments[2]} ]]
    then
        displayErrorMsgAndExit "Ooops🤔: The incremental value of the of 'Argument
2(New Queue)'
        should be greater than the incremental value of 'argument 3(Accepted
Queue)'"
else
    incValNewQ=${progArguments[1]}
    incValAcceptedQ=${progArguments[2]}
fi

# 5. MAIN ALGORITHM

echo "*****************************************"
echo "THIS IS THE SELFISH ROUND ROBIN ALGORITHM"
echo "*****************************************"
echo

# prompt the user where the output of the results(of the algorithm) should be
displayed.
while true;
do
    echo -e "Which of the following options do you want the output of the results
to be displayed?  [OPTIONS: STDOUT or FILE or BOTH]"
    read userFeedback

    # Convert userFeedback to lowercase (for case-insensitive comparison)
    directOutput=$(echo "$userFeedback" | tr '[:upper:]' '[:lower:]')

    if [[ "$directOutput" == "stdout" || "$directOutput" == "file" ||
"$directOutput" == "both" ]]; then
        # if a correct option is given, exit the loop.
        if [[ $directOutput == "file" || $directOutput == "both"  ]]
        then
            while true;
            do
                echo -e "Which filename do you want to store output of the
results?"
                read userFeedback
                if [[ -n "$userFeedback" ]]; then
                    outputToFile="$userFeedback"
                    break
                else
                    echo "CAUTION❗: Kindly enter a filename. Cannot leave this
field
                    blank."
                fi
            done
```

```
        fi
        break
    else
        echo "CAUTION!: $userFeedback is not a valid option. Please provide either
        STDOUT or FILE or BOTH."
    fi
done

echo

# Retrieve process from the provided path
while read -r line || [[ -n $line ]];
do
  read -r name service arrival <<<"$line"

  # Appending the new process array to the existing process array
  nameOfProcesses+=("$name")
  serviceTimeProcesses+=("$service")
  arrivalTimeProcesses+=("$arrival")
done < "${progArguments[0]}"

# Printing out the header for the program(prints the process's names ontop of the
# result's output)
headers="Time       "
for name in "${nameOfProcesses[@]}";
do
    headers+=$(createHeader "$name" 10)
done

outputResultDestination "$directOutput" "$headers" "$outputToFile"

while [ $processesCompleted -lt ${#nameOfProcesses[@]} ]
do

    if [ ${#nameOfAcceptedQ[@]} -gt 0 ];
    then
        # a. Shifting or moving the first process to the back or end of the queue.
        if [ $timeConsumed -eq $quanta ]
        then
            timeConsumed=0
            firstProcessName=${nameOfAcceptedQ[0]}
            firstProcessService=${serviceTimeAcceptedQ[0]}
            firstProcessPriority=${priorityAcceptedQ[0]}
            firstProcessStatus=${statusAcceptedQ[0]}

            #Error.  so change
            if [[ $firstProcessStatus == "R" ]]
            then
            nameOfAcceptedQ=("${nameOfAcceptedQ[@]:1}" "$firstProcessName")
            serviceTimeAcceptedQ=("${serviceTimeAcceptedQ[@]:1}"
"$firstProcessService")
```

```
            priorityAcceptedQ=("${priorityAcceptedQ[@]:1}" "$firstProcessPriority")
            statusAcceptedQ=("${statusAcceptedQ[@]:1}" "$firstProcessStatus")
            fi

            # Modify the status and decrease the service time of first process
            statusAcceptedQ[0]="R"
            # Modify the previous first process status to waiting(W)
            if [ ${#nameOfAcceptedQ[@]} -gt 1 ];
            then
                statusAcceptedQ[${#statusAcceptedQ[@]} - 1]="W"
            fi
        fi

        ((serviceTimeAcceptedQ[0]--))
        verifyQuanta
    fi



    # b. Verifying that all processes have successfully been loaded
    if [ $(( ${#nameOfAcceptedQ[@]} + ${#nameOfNewQ[@]} + $processesCompleted )) -
ne ${#nameOfProcesses[@]} ]; then

        # Retrieving processes based on their their time of arrival.
        for ((i = 0; i < ${#arrivalTimeProcesses[@]}; i++));
        do
            if [ "${arrivalTimeProcesses[i]}" -eq $time ]; then
                # Placing the process into the new or accepted queue depending on
                # the presence of a process in the accepted queue.
                if [ ${#nameOfAcceptedQ[@]} -eq 0 ]; then
                    nameOfAcceptedQ+=("${nameOfProcesses[i]}")
                    serviceTimeAcceptedQ+=("${serviceTimeProcesses[i]}")
                    priorityAcceptedQ+=(0)
                    statusAcceptedQ+=("R")

                    # Decrease the time of service of the first process.
                    ((serviceTimeAcceptedQ[0]--))
                    verifyQuanta
                else
                    nameOfNewQ+=("${nameOfProcesses[i]}")
                    serviceTimeNewQ+=("${serviceTimeProcesses[i]}")
                    priorityNewQ+=(0)
                    statusNewQ+=("W")
                fi

            fi


        done

    fi
```

```
    # c.increasing the priority of new and accepted queues based on their
respective
    # incremental values
    for ((i = 0; i < ${#priorityAcceptedQ[@]}; i++)); do
        ((priorityAcceptedQ[i]+=incValAcceptedQ))
    done

    for ((i = 0; i < ${#priorityNewQ[@]}; i++)); do
        ((priorityNewQ[i]+=incValNewQ))
    done

    # c. Displaying the current status of all processses.
    adjustedNumSpaces=$(adjustNumSpaces 10 "$time")
    readingData=$(createHeader "$time" "$adjustedNumSpaces")
    for index in "${!nameOfProcesses[@]}";
    do
        name="${nameOfProcesses[index]}"
        arrival="${arrivalTimeProcesses[index]}"

        s=
        for ((i = 0; i < ${#nameOfAcceptedQ[@]}; i++));
        do
            if [ "${nameOfAcceptedQ[i]}" == "$name" ];
            then
                s+="${statusAcceptedQ[i]}"
            fi
        done

        if [ -z "$s" ]
        then
            for ((i = 0; i < ${#nameOfNewQ[@]}; i++));
            do
                if [ "${nameOfNewQ[i]}" == "$name" ];
                then
                    s+="${statusNewQ[i]}"
                fi
            done
        fi

        if [ -n "$s" ];
        then
            readingData+=$(createHeader "$s" 10)
        elif [ $(( $arrival )) -le ${time} ]; then
            readingData+=$(createHeader "F" 10)
        else
            readingData+=$(createHeader "-" 10)
        fi

    done
```

```
    outputResultDestination "$directOutput" "$readingData" "$outputToFile"
"addParameter"

    # d. Verifying if any process has completed its execution(Service time-NUT) and
    #remove them from the accepted queue.
    previousAcceptedQLength=${#nameOfAcceptedQ[@]}
    tempAcceptedQueueName=()
    tempAcceptedQueueService=()
    tempAcceptedQueuePriority=()
    tempAcceptedQueueStatus=()

    for ((i = 0; i < ${#serviceTimeAcceptedQ[@]}; i++));
    do
        if [ -n "${serviceTimeAcceptedQ[i]}" ] && [ "${serviceTimeAcceptedQ[i]}" -
gt 0 ];
        then
            tempAcceptedQueueName+=("${nameOfAcceptedQ[i]}")
            tempAcceptedQueueService+=("${serviceTimeAcceptedQ[i]}")
            tempAcceptedQueuePriority+=("${priorityAcceptedQ[i]}")
            tempAcceptedQueueStatus+=("${statusAcceptedQ[i]}")
        fi
    done

    # e. Increase the count of completed processes according to the number of
    # filtered-out processes.
    diff=$((previousAcceptedQLength - ${#tempAcceptedQueueName[@]}))
    ((processesCompleted += diff))

    nameOfAcceptedQ=("${tempAcceptedQueueName[@]}")
    serviceTimeAcceptedQ=("${tempAcceptedQueueService[@]}")
    priorityAcceptedQ=("${tempAcceptedQueuePriority[@]}")
    statusAcceptedQ=("${tempAcceptedQueueStatus[@]}")

    # f. Verify and eliminate/remove processes from the new queue when their
priority
    # matches that of the accepted queue.
    tempNewQueueName=()
    tempNewQueueService=()
    tempNewQueuePriority=()
    tempNewQueueStatus=()
    for ((i = 0; i < ${#nameOfNewQ[@]}; i++));
    do
        firstPriority="${priorityAcceptedQ[0]}"
        if [ ! -n "$firstPriority" ]; then
            firstPriority=0
        fi


        # To examine whether the priority of element in the new queue is greater
        # than or equal to the priorty of the first element in the accepted queue
        if [ "${priorityNewQ[i]}" -ge "$firstPriority" ];
```

```
        then
            nameOfAcceptedQ+=("${nameOfNewQ[i]}")
            serviceTimeAcceptedQ+=("${serviceTimeNewQ[i]}")
            priorityAcceptedQ+=("${priorityNewQ[i]}")
            statusAcceptedQ+=("${statusNewQ[i]}")

        else
            tempNewQueueName+=("${nameOfNewQ[i]}")
            tempNewQueueService+=("${serviceTimeNewQ[i]}")
            tempNewQueuePriority+=("${priorityNewQ[i]}")
            tempNewQueueStatus+=("${statusNewQ[i]}")
        fi
    done

    nameOfNewQ=("${tempNewQueueName[@]}")
    serviceTimeNewQ=("${tempNewQueueService[@]}")
    priorityNewQ=("${tempNewQueuePriority[@]}")
    statusNewQ=("${tempNewQueueStatus[@]}")

    ((time++))
done

adjustedNumSpaces=$(adjustNumSpaces 10 "$time")
footer=$(createHeader "$time" "$adjustedNumSpaces")
for name in "${nameOfProcesses[@]}"; do
  footer+="F          "
done

outputResultDestination "$directOutput" "$footer" "$outputToFile" "addParameter"

exit 0
```

LINK TO VIDEO OF TESTING THE SRR ALGORITHM SCRIPT

https://youtu.be/9GNAf3L6--E?si=nvKGua9iheztsYx5

REFERENCES

1. Geeksforgeeks(2018). “Selfish Round-Robin CPU Algorithm.”Available at: https://www.geeksforgeeks.org/selfish-round-robin-cpu-scheduling/ (Accessed: 01/02/2023).

2. Pranavnath(18/07/2023). “Selfish Round-Robin CPU Scheduling. ”Available athttps://www.tutorialspoint.com/selfish-round-robin-cpu-scheduling/ (Accessed: 02/12/2023).

3. HindiCodingCommunity(26/02/2023).” Selfish Round-Robin CPU Scheduling in Operating systems.(online) ”Available at: https://www.hindicodingcommunity.com/2023/02/selfish-round-robin-cpu-scheduling-in.html/ (Accessed: 02/12/2023).

4. Tutorialspoint (n.d). “Selfish Round-Robin CPU Scheduling. (online)”Available athttps://www.tutorialspoint.com/selfish-round-robin-cpu-scheduling/ (Accessed: 02/12/2023).