Muhammed Sahil Arayakandy
Instructor: Prof. George C.
13 December 2023

Software Development Environments

Coursework

Document Outline

Introduction

In this project, we are going to write a program to simulate the Selfish Round Robin CPU Scheduling Algorithm in Bash.

Selfish Round Robin (SRR)

The Selfish Round Robin (SRR) scheduling algorithm is a modification of the traditional Round Robin algorithm. The objective of SRR is to give better service to processes that have been executing for a while than to newcomers. In SRR, processes in the ready list are partitioned into two lists: NEW and ACCEPTED. The New processes wait while Accepted processes are serviced by the Round Robin. The priority of a new process increases at the rate ‘a’ while the priority of an accepted process increases at the rate ‘b’.

Bash

Bash is the GNU Project's shell — the Bourne Again Shell. This is an sh-compatible shell that incorporates useful features from the Korn shell (ksh) and the C shell (csh). It is intended to conform to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard.

Requirements

Functional Requirements:

- The bash script should emulate the behaviour of Selfish Round Robin Scheduling Algorithm (SRR).
- The script should accept 3 positional parameters:
    - 1st: Data file name
    - 2nd: The increment integer value of new queue
    - 3rd: The increment integer value of accepted queue
- The script should also accept a fourth optional parameter: the quanta number.
- For output, the script should offer the user 3 choices:
    - Output to standard output only
    - Output to a named text file (if the file exists then it should be overwritten)
    - Output to both standard output and named text file
- The script should read the data and then emulate and output the behaviour of the algorithm showing for each time interval of 1 quanta, which process is running (**R**), waiting (**W**), completed (**F**) or not arrived (**-**), with respect to each time step.
- The script should terminate when all processes complete.

External Interface Requirements:

- The format of the named data file provided by the user should contain the processes label, the NUT Value and the arrival time for each processes in that specific order for each process. Eg. Process A with NUT 6 arriving at T=0 is 'A 6 0'
- Each process is separated by a newline in the input file
- The input file must be a regular file
- The 2nd, 3rd & 4th positional parameters must be numbers.

Non-Functional Requirements:

- The script should ensure correct formatting of output for improved readability.

Design of System

Overall Architecture

- All processes are converted to jobs that are stored as strings with each character in the string denoting various aspects of the process separated by spaces. For eg: 'A 6 0 2 W' is referring to process A, with NUT value of 6, arriving at T=0, having a priority of 2 and a W (Waiting) status, respectively.
- We use 3 bash arrays to represent the New Queue - 'newQ', the Accepted Queue - 'accQ' and the Finished Queue - 'finishedQ'. 'newQ' and 'accQ' are used for the Selfish Round Robin Algorithm (SRR) and 'finishedQ' is used to store the jobs that have completed.
- We also have variable 'a' = priority of new queue, variable 'b' = priority of accepted queue and variable 'q' = quanta value.
- We use an infinite while loop tracking a variable 't', starting at 0, to represent the flow of time.
- We use a loop exit condition that becomes to true when all jobs are completed to break out of the loop and exit the script.

Validation Checks

- Ensure the number of parameters entered by the user are equal 3 or 4
- Ensure the 2nd, 3rd & 4th parameters are numbers
- Ensure the input file entered is a regular file

Data Pre-Processing

- Read the data from the input file and create the appropriate data structures, I.e, arrays of strings.
- Sort all the processes according to their arrival time (AT)
- Prompt the user to get their output choice, I.e if they would like to output the statuses to either standard output, a named file or to both.

Core Algorithm

1. Start the time loop, with t=0.
2. When t == AT of a processes, add it as a job to the accepted queue if both accepted queue and new queue are empty, otherwise add it to the new queue.
3. All new jobs start with priority = 0 and status as '-'.
4. Loop over every job in both new and accepted queues and change the status to 'W'. This ensure that processes that are in 'R' from previous runs of the loop do not cause any problems in the output.
5. Now, set the top job,, i.e job at index 0 of accepted queue to 'Run'. So, change its status to 'R' and decrement the NUT by quanta value.
6. Increment the value of all Jobs in both new queue and accepted queue according to the corresponding values inputted by the user, i.e, 'a' & 'b', respectively.
7. Find the lowest priority thats on the accepted queue and check to see if there are any jobs on the new queue that have the same or higher priority. If there are, move them over to the accepted queue.
8. Now apply the SRR algorithm. If the top job, I.e, the job at accepted queue index 0 does not have a NUT value of 0, then move to the back of the accepted queue.
9. Create the output string that show the status of each process along with the time and output it to either standard output, a file or both based on the choice the user selected.
10. Check for the loop exit condition, ie if the number of jobs in finishedQ is the same as the number of initial processes.
11. If the NUT value of the top job, I.e the job at accepted queue index 0, is less than or equal to zero, then move this job over to the finished queue.
12. Finally, we increment 't' by +1.
13. Inform the user of program competition on breaking out of the time loop and exit the script.

Implementation

```
#!/bin/bash
#
# Simulate the behaviour of Selfish Round Robin Scheduling
Algorithm.
# Author: Muhammed Sahil Arayakandy
#

input=$1 # data file entered
a=$2 # priority increment for new queue
b=$3 # priority increment for accepted queue

q=1 # default quanta value

# Check number of parameters
if [[ $# -eq 3 ]]
then
  echo "Right number of parameters entered: $#"
elif [[ $# -eq 4 ]]
then
  echo "Right number of parameters entered: $#"
  q=$4 # assign user defined quanta value
else
  echo "Error! Incorrect number of parameters entered: $# (it
should be 3 or 4)"
  exit
fi

# Check that the 2nd, 3rd & 4th parameters are numbers
str="$2$3$4"
nchar=${#str}

i=0
while [[ $i -lt $nchar ]]
do
    ch=${str:i:1}

    case $ch in
    0|1|2|3|4|5|6|7|8|9) numf=1
                        ;;
    *) numf=0
        break
        ;;
    esac

    ((i = $i + 1))
done
```

```
if test $numf = 1
then
    echo "digit match"
else
    echo "Error! 2nd, 3rd & 4th (Optional) Parameters have to be a
number"
    exit
fi

# Check that input file is a regular file
if [[ -f $input ]]
then
  echo "$input data file entered"
else
  echo "Error! Input file needs to be a regular file"
  exit
fi

# Prompt the user for output choice
echo "*****************************"
echo "Please choose output option"
echo "*****************************"
echo "1 ... Standard Output Only"
echo "2 ... Named Text File Only"
echo "3 ... Both"
echo "****************************"

read -p "option : " option
echo "****************************"

if [[ $option -eq 1 ]]
then
  echo "Chosen Option: Standard Output Only"
elif [[ $option -eq 2 ]]
then
  echo "Chosen Option: Named Text File Only"
  read -p "Please enter the output file name: " fname
elif [[ $option -eq 3 ]]
then
  echo "Chosen Option: Both"
  read -p "Please enter the output file name: " fname
else
  echo "Invalid input for Option. Please enter a valid number."
  exit
fi
```

```
processes=() # raw processes

# read the input and build the processes array
while read -r id nut at
do
  processes[${#processes[@]}]="$id $nut $at"
done < $input

# print each item in array
for process in "${processes[@]}"
do
     echo "$process"
done
echo "Priority Increment in New_Queue = $a and in Accepted_Queue =
$b”

#######################################
# Outputs the header to stdout
# Globals:
#   ${processes[@]}
# Arguments:
#   None
# Outputs:
#   Writes header to stdout
#######################################
function outputHeaderToStdOut {
  echo -e "T \t\c"
  for process in "${processes[@]}"
  do
    p=($process)
    echo -e "${p[0]}  \t\c"
  done
  echo -e "\n"
}

#######################################
# Outputs the header to file.
# Globals:
#   ${processes[@]}
# Arguments:
#   None
# Outputs:
#   Writes header to file
#######################################
function outputHeaderToFile {
  echo -e "T \t\c" > $fname
  for process in "${processes[@]}"
  do
    p=($process)
    echo -e "${p[0]}  \t\c" >> $fname
  done
  echo -e "\n" >> $fname
}
```

```
# Print the Headers for output
if [[ $option -eq 1 ]]
then
  outputHeaderToStdOut
elif [[ $option -eq 2 ]]
then
  outputHeaderToFile
elif [[ $option -eq 3 ]]
then
  outputHeaderToStdOut
  outputHeaderToFile
fi

# sort processes according to arrival time
i=0
nelem=${#processes[@]}

while [[ $i -lt $nelem ]]
do
  j=0
  while [[ $j -lt $nelem ]]
  do
    p1=(${processes[j]})
    at1=${p1[1]}

    p2=(${processes[j+1]})
    at2=${p2[1]}

    if [[ $at1 -lt $at2 ]]
    then
      processes[j]="${p2[@]}"
      processes[j+1]="${p1[@]}"
    fi

   ((j = $j + 1))
  done

  ((i = $i + 1))
done


# Core Algorithm

t=0 # time

newQ=() # the new queue
accQ=() # the accepted queue
finishedQ=() # the finished queue
```

```
while [[ true ]]
do
  # Add jobs based on the arrival time
  for process in "${processes[@]}"
  do
  p=($process)

  if [[ ${p[2]} -eq $t ]]
  then
      if [[ ${#newQ[@]} -eq 0 && ${#accQ[@]} -eq 0 ]]
      then
      accQ[${#accQ[@]}]="${p[0]} ${p[1]} ${p[2]} 0 -"
      else
      newQ[${#newQ[@]}]="${p[0]} ${p[1]} ${p[2]} 0 -"
      fi
  fi
  done

  # Make the status's of all jobs to 'W' (Waiting)
  for (( i = 0; i < ${#newQ[@]}; i++ ))
  do
    job=${newQ[i]}
    j=($job)

    j[4]="W"

    newQ[i]="${j[@]}"
  done

  for (( i = 0; i < ${#accQ[@]}; i++ ))
  do
    job=${accQ[i]}
    j=($job)

    j[4]="W"

    accQ[i]="${j[@]}"
  done
  # Set top job to 'R' and decrement the NUT
  if [[ ${#accQ[@]} -ne 0 ]]
  then
    job=${accQ[0]}
    j=($job)

    j[4]="R"

    nut=${j[1]}
    (( nut = $nut - 1 ))
    j[1]=$nut

    accQ[0]="${j[@]}"
  fi
```

```
# Increment the priority of all jobs with user defined values
for (( i = 0; i < ${#newQ[@]}; i++ ))
do
  job=${newQ[i]}
  j=($job)

  pr=${j[3]}
  (( pr = $pr + $a ))
  j[3]=$pr

  newQ[i]="${j[@]}"
done

for (( i = 0; i < ${#accQ[@]}; i++ ))
do
  job=${accQ[i]}
  j=($job)

  pr=${j[3]}
  (( pr = $pr + $b ))
  j[3]=$pr

  accQ[i]="${j[@]}"
done

# Move job on priority match
accQPriorities=()
lowestAccQPriority=0

for (( i = 0; i < ${#accQ[@]}; i++ ))
do
  job=${accQ[i]}
  j=($job)

  pr=${j[3]}
  accQPriorities+=($pr)
done

# get the lowest priority in accepted queue
readarray -td '' sorted < <(printf '%s\0' "${accQPriorities[@]}"
| sort -z)
lowestAccQPriority=${sorted[0]}
```

```
# find and move the jobs appropriately
for (( i = 0; i < ${#newQ[@]}; i++ ))
do
  job=${newQ[i]}
  j=($job)

  pr=${j[3]}

  if [[ $pr -ge $lowestAccQPriority ]]
  then
    unset newQ[$i]

    accQ[${#accQ[@]}]="${j[@]}"
  fi
done

newQ=("${newQ[@]}") # reindex the array after using unset

# Apply Selfish Round Robin
if [[ ${#accQ[@]} -ne 0 ]]
then
  job=${accQ[0]}
  j=($job)

  nut=${j[1]}

  if [[ $nut -ne 0 ]]
  then
    unset accQ[0]
    accQ=("${accQ[@]}") # reindex the array after using unset

    accQ[${#accQ[@]}]="${j[@]}"
  fi
fi
```

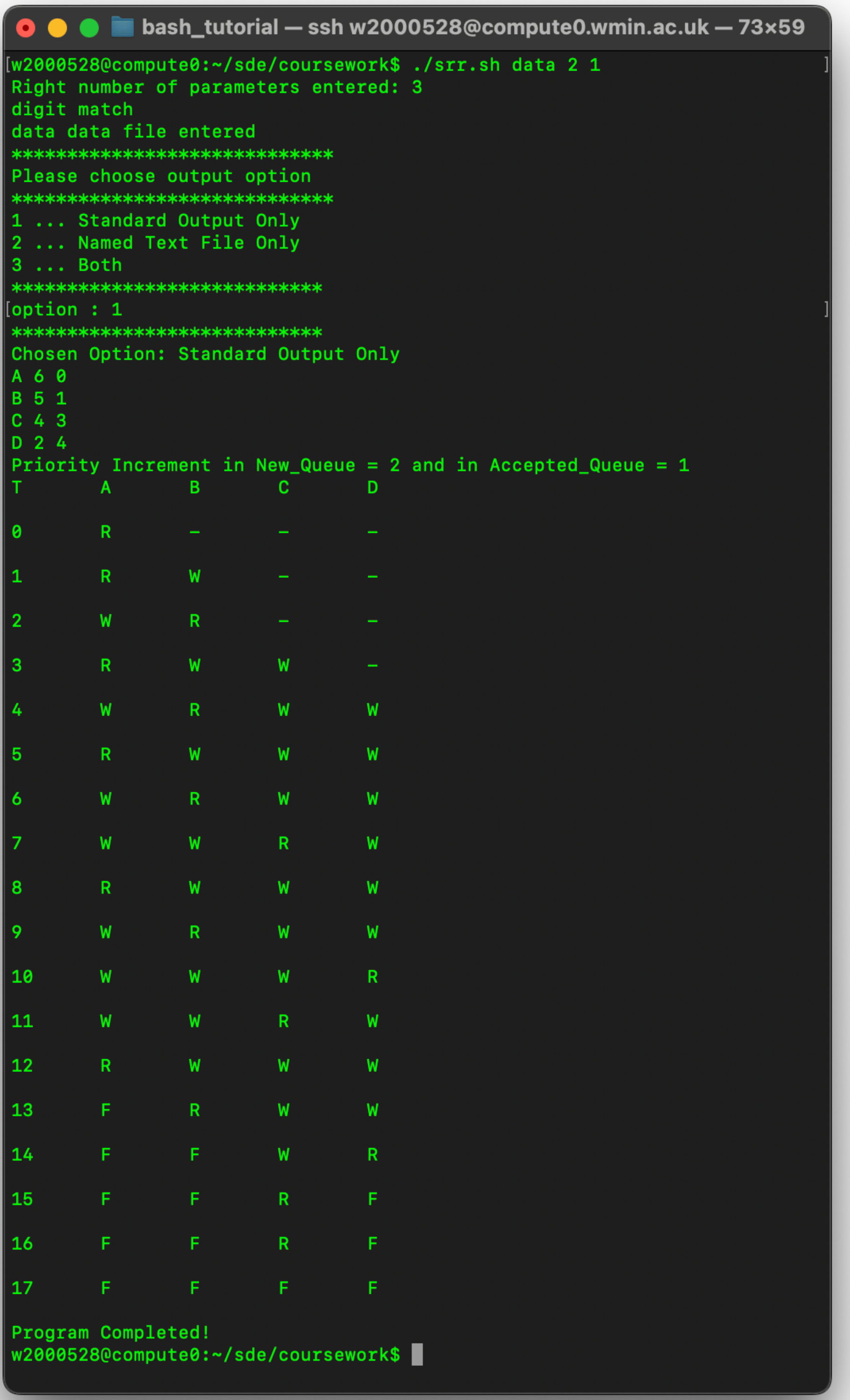
```
# Print the Statuses
statuses=()
allJobs=( "${newQ[@]}" "${accQ[@]}" "${finishedQ[@]}" )

for process in "${processes[@]}"
do
  p=($process)
  pID=${p[0]}

  status="-"

  for job in "${allJobs[@]}"
  do
    j=($job)
    jID=${j[0]}

    if [[ "$pID" == "$jID" ]]
    then
      status=${j[4]}
    fi
  done

  statuses[${#statuses[@]}]=$status
done

#######################################
# Outputs the statuses to stdout
# Globals:
#   ${statuses[@]}
#   $t
# Arguments:
#   None
# Outputs:
#   Writes time and statuses to stdout
#######################################
function outputStatusesToStdOut {
  echo -e "$t \t\c"

  for status in "${statuses[@]}"
  do
    echo -e "$status  \t\c"
  done

  echo -e "\n"
}
```

```
#######################################
# Outputs the statuses to file
# Globals:
#   ${statuses[@]}
#   $t
# Arguments:
#   None
# Outputs:
#   Writes time and statuses to file
#######################################
function outputStatusesToFile {
  echo -e "$t \t\c" >> $fname

  for status in "${statuses[@]}"
  do
    echo -e "$status  \t\c"  >> $fname
  done

  echo -e "\n"  >> $fname
}

if [[ $option -eq 1 ]]
then
  outputStatusesToStdOut
elif [[ $option -eq 2 ]]
then
  outputStatusesToFile
elif [[ $option -eq 3 ]]
then
  outputStatusesToStdOut
  outputStatusesToFile
fi

# Exit loop if all jobs have finished
if [[ ${#finishedQ[@]} -eq ${#processes[@]} ]]
then
  break
fi
```

```
  # Move finished jobs to finished queue
  job=${accQ[0]}
  j=($job)

  nut=${j[1]}
  if [[ $nut -le 0 ]]
  then
    unset accQ[0]
    accQ=("${accQ[@]}")

    j[4]="F"
    finishedQ[${#finishedQ[@]}]="${j[@]}"
  fi

  # Increment t by 1
  (( t = $t + 1 ))
done

echo "Program Completed!"
exit
```

Link To Video:

https://www.loom.com/share/c3d5cac298a2455f8ad393a6e28bf4eb?sid=5d55798c-e1c3-4c20-ae3a-f6a481a3d691



bash_tutorial — ssh w2000528@compute0.wmin.ac.uk — 73×59

```
w2000528@compute0:~/sde/coursework$ ./srr.sh data 2 1
Right number of parameters entered: 3
digit match
data data file entered
*****************************
Please choose output option
*****************************
1 ... Standard Output Only
2 ... Named Text File Only
3 ... Both
****************************
option : 1
****************************
Chosen Option: Standard Output Only
A 6 0
B 5 1
C 4 3
D 2 4
Priority Increment in New_Queue = 2 and in Accepted_Queue = 1
T       A       B       C       D

0       R       -       -       -

1       R       W       -       -

2       W       R       -       -

3       R       W       W       -

4       W       R       W       W

5       R       W       W       W

6       W       R       W       W

7       W       W       R       W

8       R       W       W       W

9       W       R       W       W

10      W       W       W       R

11      W       W       R       W

12      R       W       W       W

13      F       R       W       W

14      F       F       W       R

15      F       F       R       F

16      F       F       R       F

17      F       F       F       F

Program Completed!
w2000528@compute0:~/sde/coursework$ 
```

Testing

For this testing environment, our script is called srr.sh and data file is called data.

Input Options

Test Case	Expected	Output Observed
./srr.sh 2 1	Parameter count error	Error! Incorrect number of parameters entered: 2 (it should be 3 or 4)
./srr.sh data 2 a	Invalid parameter error	Right number of parameters entered: 3 Error! 2nd, 3rd & 4th (Optional) Parameters have to be a number
./srr.sh test_dir 2 1	Input file error	Error! Input file needs to be a regular file
./srr.sh test 2 1	Successful execution	Right number of parameters entered: 3 digit match data data file entered

Algorithm

Testing the algorithm output for the following input data:

A 6 0

B 5 1

C 4 3

D 2 4

New queue priority = 2

Accepted queue priority = 1

T	Expected	Output Observed
0	R - - -	R - - -
1	R W - -	R W - -
2	W R - -	W R - -
3	R W W -	R W W -
4	W R W W	W R W W
5	R W W W	R W W W
6	W R W W	W R W W
7	W W R W	W W R W
8	R W W W	R W W W
9	W R W W	W R W W
10	W W W R	W W W R
11	W W R W	W W R W
12	R W W W	R W W W
13	F R W W	F R W W
14	F F W R	F F W R
15	F F R F	F F R F
16	F F R F	F F R F
17	F F F F	F F F F

Output Options

On manual testing of the 3 options - output to standard output, named file and both, I can confirm that all 3 executed correctly.

- Option 1: Successfully output to stdout, no file created.
- Option 2: Successfully output to the given file, no stdout output observed. If file exists, successfully overwrites the file.
- Option 3: Successfully outputs to stdout as well as to file.

Critical Evaluation

The end program meets the requirements mentioned.

Metrics for Dataset:

A 6 0

B 5 1

C 4 3

D 2 4

New queue priority = 2

Accepted queue priority = 1

Input Validation - Passed

Algorithmic Statuses - Passed

Output Options - Passed

Execution Time: 0m2.186s

However, improvements could be made to increase the robustness, efficiency and scalability of the code.

Performance

- Currently using a bubble sort algorithm to sort the processes. As this algorithm has a time complexity of O(n2), it is inefficient and will not scale to a large input. So, it could be replaced with a more efficient algorithm like quick sort.

Robustness

- Currently we are only validating the user inputs, not the data in the data file. As we have mentioned a specification for the data file, we let the user assume the responsibility of ensuring the correctness of the data file. However, we could improve the robustness of our program by adding validation for the data as well.

References

- Kowalczyk, Krzysztof (no date) *Essential Bash*. Available at: https://www.programming-books.io/essential/bash/index.html (Accessed: 13 December 2023).
- Krüger, Gerhard & Lane, Charles (2023) *How to Write a Software Requirements Specification (SRS Document).* Available at: https://www.perforce.com/blog/alm/how-write-software-requirements-specification-srs-document (Accessed: 13 December 2023)
- The Free Software Foundation (no date) *GNU Bash*. Available at: https://www.gnu.org/software/bash/ (Accessed: 13 December 2023)
- Sanchhaya Education Private Limited (2023) *Selfish Round Robin CPU Scheduling.* Available at: https://www.geeksforgeeks.org/selfish-round-robin-cpu-scheduling/ (Accessed: 13 December 2023)
- Google LLC (no date) *Shell Style Guide.* Available at: https://google.github.io/styleguide/shellguide.html (Accessed: 13 December 2023)