OSP2 FINAL PROJECT REPORT

Group Number: 4

Group Members

Rakshithkumar Narayanappa (SUID – 387764067)

Bobby Jasuja (SUID – 504488701)

Sweety NareshKumar Patel (SUID – 223416858)

Mona Ramesh (SUID – 264581836)

GOALS ACCOMPALISHED

We have completed all three goals mentioned in our project proposal:

LEVEL-1 GOALS

- ✓ Design, implementation and testing Management of tasks.
- ✓ Implementation and testing of FCFS, Round Robin algorithms and priority-driven pre-emptive scheduling - Threads.
- ✓ Implementation and testing of Page replacement algorithms - First in first out (FIFO) and least recently used (LRU). – Virtual Memory Management

LEVEL-2 GOALS

- ✓ Implementation of file systems.
- ✓ Implementation and testing of disk request scheduling.

LEVEL-3 GOALS

- ✓ Implementation of resource management module.
- ✓ Implementation and testing of inter-process communication using ports
- ✓ Implementation of scheduling algorithms with enhancement and better performance on comparison with FCFS, round robin and priority-driven pre-emptive scheduling- (**Multi-Level Feedback Queue)**
- ✓ Implementation of page replacement algorithm more efficient that FIFO- (**Second Chance)**

PREFACE

OSP2 is both an implementation of a modern operating system, and a flexible environment for generating implementation projects appropriate for an introductory course in operating system design.

OSP2 is written in the Java programming language and students program their OSP2 projects in Java as well.

OSP2 consists of number of modules, each of which performs a basic operating systems service, such as device scheduling, CPU scheduling, interrupt handling, file management, memory management, process management, resource management, and inter-process communication. The OSP2 distribution comes with a reference Java implementation of each module.

The heart of OSP2 is a simulator that gives the illusion of a computer system with a dynamically evolving collection of user processes to be multi-programmed. All the other modules of OSP2 are built to respond appropriately to the simulator-generated events that drive the operating system. The simulator "understands" its interaction with the other modules in that it can often detect an erroneous response by a module to a simulated event. In such cases, the simulator will gracefully terminate execution of the program by delivering a meaningful error message to the user, indicating where the error might be found. This facility serves both as a debugging tool for the student and as teaching tool for the instructor, as it ensures that student programs acceptable to the simulator are virtually bug-free.

The difficulty of the job streams generated by the simulator can be dynamically adjusted by manipulating the simulation parameters. This yields a simple and effective way of testing the quality of student programs.

There is also a layer that sits between the event engine and the package layer, the so-called Interface Layer or IFL. The IFL monitors the execution of the packages for the purpose of catching semantic errors, and for gathering performance statistics. Thus, the IFL can be viewed as a protective "wrapper" around the packages.

Simulated Hardware in OSP2 The Hardware and the Interrupts packages of OSP2 model the hardware-oriented aspects of the simulated multiprogramming operating system. Hardware consists of four Java classes, which we are:

- CPU
- Disk
- HCLOCK
- Interrupt Vector

The utilities package contains number of classes that are needed purely for simulation support. It also provides a class, Global Variables, that is required by the packages, and several other "utility" classes that assist in implementing the projects.

INTRODUCTION-

OSP 2 incorporates various operating system modules. Since it is implemented in Java, it is highly modular and each module can have their own implementations. Each module performs it basic functionality. We have implemented following seven modules in our OSP2 project -

- **Management of tasks**
- **Management and scheduling of threads**
- **Virtual memory management**
- **File system**
- **Scheduling of disks(Device)**
- **Resource Management**
- **Inter-process communication(Port)**

PROJECT IMPLEMENTATION-

The Detailed description of the implementation of these modules and the results observed are explained below: -

Management of tasks:

This module deals with the creation and deletion of tasks and associated methods like creation and deletion of threads, creation and deletion of ports associated with the task, addition, and removal of files to the list of files associated with the task.

Files	Implementation
TaskCB.java	Used ArrayList to manage threads, ports and files which are associated with a task Created the task after associating the page table entry, setting its creation time, status, priority and allocating swap file. Deletion of the task after removing all ports, files and threads associated with it.

Snapshot-

```
TASKS and THREADS:
  CPU Utilization: 74.1308%
  Average service time per thread: 24561.396
  Average normalized service time per thread: 0.060128793
  Total number of tasks: 3
  Running thread(s): none
  Threads summary: 15 alive
  Among live threads: 0 running
                      15 suspended
                      0 ready
  ready queue = ()
  running thread(s) = ()
  waiting thread(s) = (132:12,149:12,158:12,159:12,160:14,164:14,166:15,167:12,168:12,169:12,170:12,171:12,172:14,173:14,174:12)
  thread(s) in pagefault = (149:12,158:12,159:12,160:14,164:14,166:15,168:12,169:12,172:14,173:14,174:12)
  killed thread(s) = (Thread(2:1/KL),Thread(8:1/KL),Thread(0:1/KL),Thread(1:1/KL),Thread(3:1/KL),Thread(4:1/KL),Thread(5:1/KL),Thread(6:1/KL),Thread(7:1/KL),
  Thread(9:1/KL),Thread(10:1/KL),Thread(12:1/KL),Thread(13:1/KL),Thread(15:1/KL),Thread(19:1/KL),Thread(20:1/KL),Thread(22:1/KL),Thread(30:6/KL),Thread(17:4/KL),
  Thread(34:7/KL),Thread(28:5/KL),Thread(35:2/KL),Thread(24:5/KL),Thread(38:2/KL),Thread(31:2/KL),Thread(42:4/KL),Thread(27:5/KL),Thread(23:2/KL),Thread(18:5/KL),
  Thread(11:2/KL),Thread(25:5/KL),Thread(29:2/KL),Thread(36:2/KL),Thread(40:2/KL),Thread(45:2/KL),Thread(46:2/KL),Thread(47:2/KL),Thread(48:2/KL),Thread(49:2/KL),
  Thread(50:2/KL),Thread(52:2/KL),Thread(54:2/KL),Thread(60:3/KL),Thread(61:6/KL),Thread(58:6/KL),Thread(14:3/KL),Thread(65:3/KL),Thread(64:3/KL),Thread(55:3/KL),
  Thread(68:6/KL),Thread(70:3/KL),Thread(57:6/KL),Thread(62:3/KL),Thread(63:3/KL),Thread(66:3/KL),Thread(69:3/KL),Thread(71:3/KL),Thread(73:3/KL),Thread(74:3/KL),
  Thread(79:3/KL),Thread(82:3/KL),Thread(83:3/KL),Thread(32:6/KL),Thread(77:6/KL),Thread(59:6/KL),Thread(51:6/KL),Thread(53:6/KL),Thread(75:6/KL),Thread(76:6/KL),
  Thread(80:6/KL),Thread(81:6/KL),Thread(84:6/KL),Thread(86:6/KL),Thread(88:6/KL),Thread(91:6/KL),Thread(93:6/KL),Thread(85:8/KL),Thread(92:9/KL),Thread(98:9/KL),
  Thread(97:9/KL),Thread(104:10/KL),Thread(99:9/KL),Thread(107:9/KL),Thread(100:9/KL),Thread(102:9/KL),Thread(112:9/KL),Thread(101:9/KL),Thread(26:4/KL),
  Thread(117:4/KL),Thread(106:9/KL),Thread(87:4/KL),Thread(89:4/KL),Thread(108:4/KL),Thread(113:4/KL),Thread(114:4/KL),Thread(116:4/KL),Thread(118:4/KL),
  Thread(119:4/KL),Thread(121:4/KL),Thread(123:4/KL),Thread(130:11/KL),Thread(94:9/KL),Thread(95:9/KL),Thread(96:9/KL),Thread(124:9/KL),Thread(126:9/KL),
  Thread(129:9/KL),Thread(136:9/KL),Thread(138:9/KL),Thread(110:11/KL),Thread(111:11/KL),Thread(125:11/KL),Thread(127:11/KL),Thread(128:11/KL),Thread(131:11/KL),
  Thread(133:11/KL),Thread(139:11/KL),Thread(140:11/KL),Thread(141:11/KL),Thread(143:11/KL),Thread(105:10/KL),Thread(148:13/KL),Thread(145:13/KL),Thread(144:10/KL),
  Thread(153:12/KL),Thread(142:10/KL),Thread(146:10/KL),Thread(147:10/KL),Thread(150:10/KL),Thread(151:10/KL),Thread(152:10/KL),Thread(154:10/KL),Thread(156:10/KL),
  Thread(157:10/KL),Thread(162:10/KL),Thread(163:10/KL))


FILES:
  Total number of files: 2
  Total number of directories: 3
  Total number of open files: 0
  Device 0: 1344 out of 1536 blocks are free, 12.5% used
  Device 1: 0 out of 40 blocks are free, 100.0% used
  Device 2: 0 out of 16 blocks are free, 100.0% used


PORTS:
  Number of Live Ports:     4
  Number of Ports Created:  23
  Number of Ports Destroyed:    19
  Number of Messages Sent:  19
```

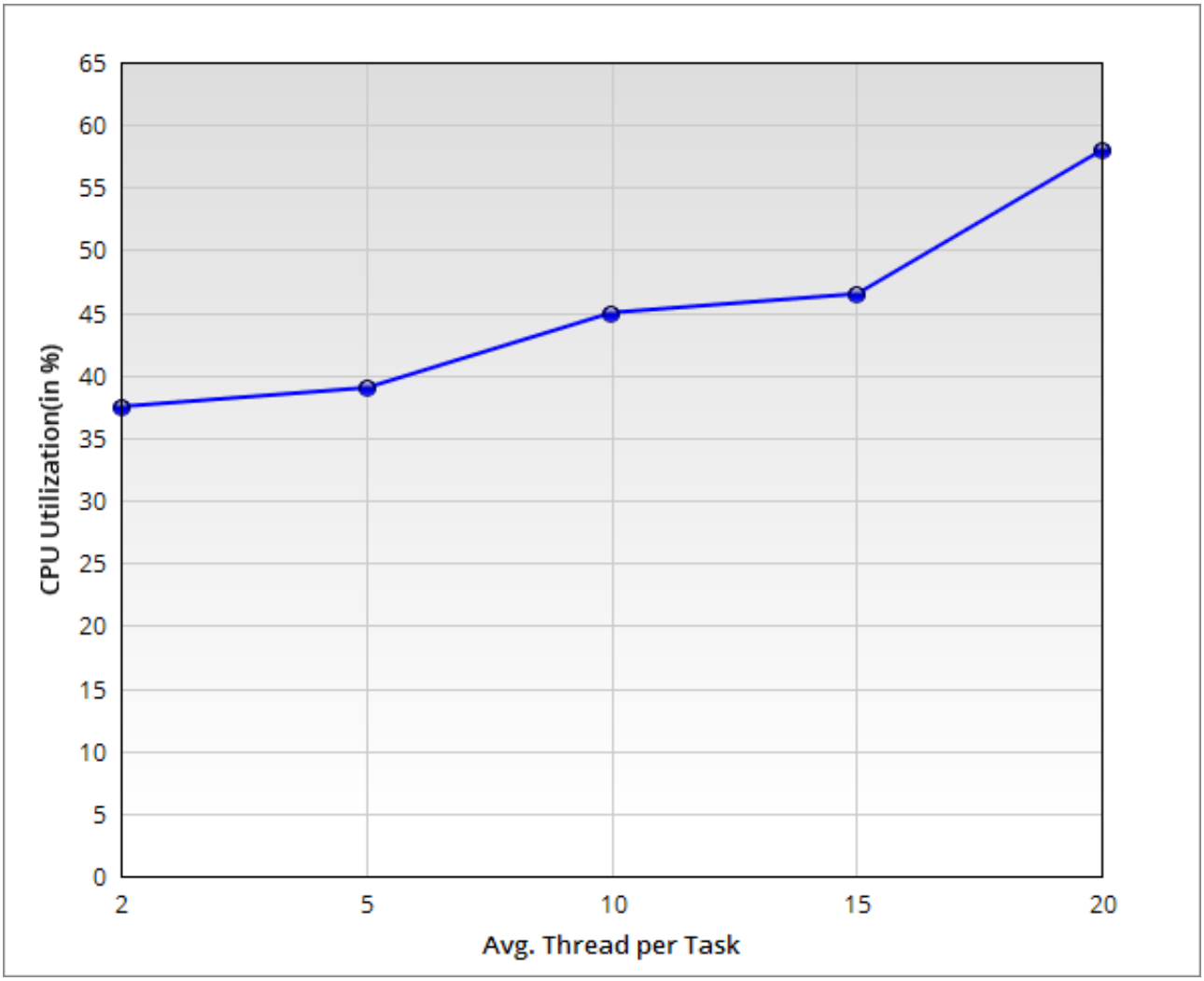
This proves that our implementation of the task module is correct since the simulation is successful and a list of tasks and associated threads, ports and files are maintained properly

Performance Analysis-

Observation 1-We used ArrayList data structure to hold resources like threads, files & ports. Tried testing with LinkedList & Vector data structure

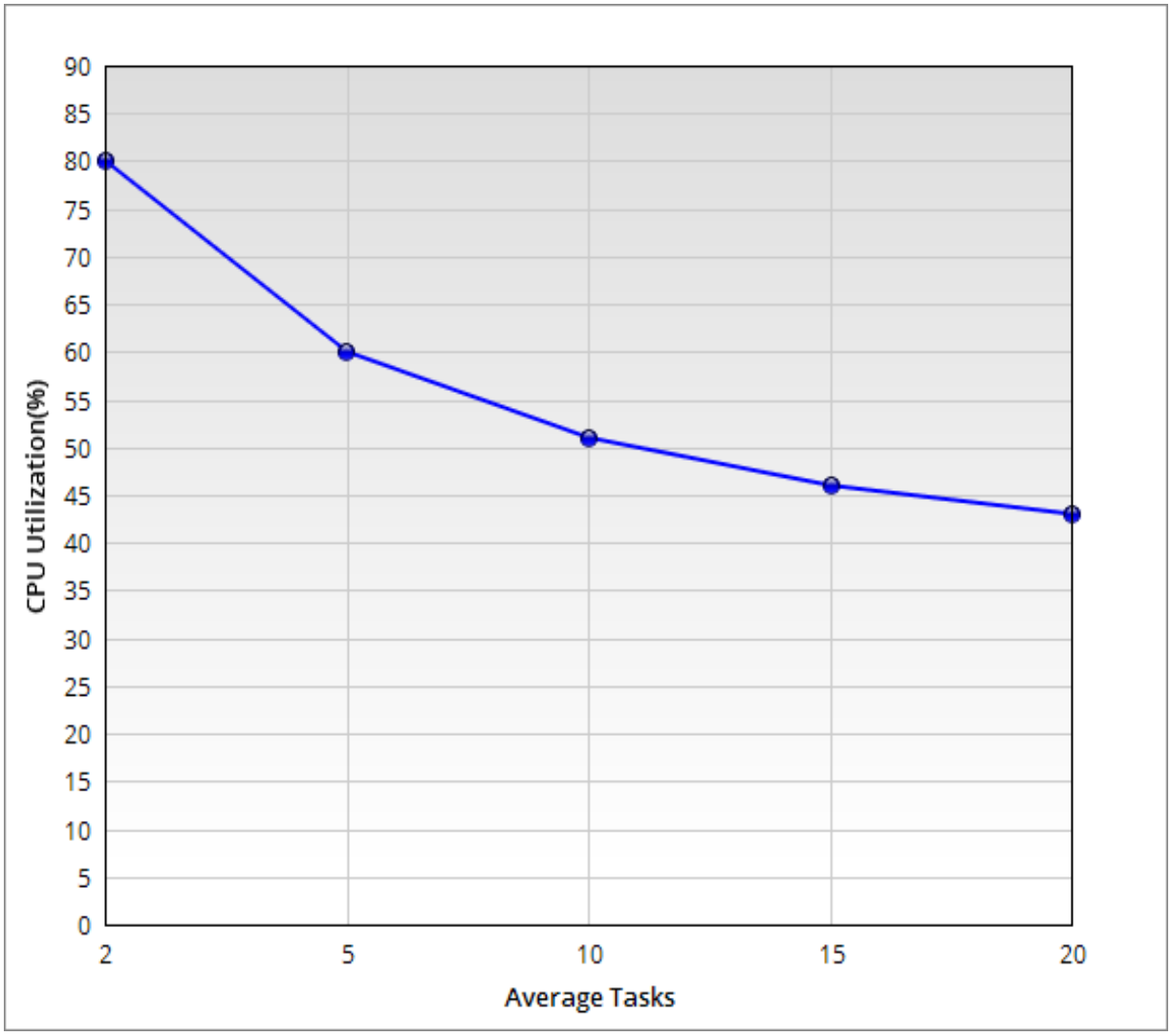
Inference- Using different data structure did not have noticeable change in CPU Utilization

Observation 2- Average threads per task vs Average CPU Utilization



Inference- CPU utilization is increased as the number of threads per task are increased

Observation 3- Average tasks vs Average CPU Utilization



Inference- CPU utilization is decreased as number tasks are increased

Management and Scheduling of Threads:

This module is responsible for creating, killing, dispatching, suspending, and resuming threads, the fundamental units of execution in OSP2**.** The thread part implements certain functions of the thread management based on Round Robin, FCFS, Priority-driven preemptive scheduling and multi-level feedback queue algorithm. We **calculated response time, throughput and turnaround time in implementation of each of these algorithms and it's present in log files.**

Performance-related measures:

1. CPU utilization: percentage of time the CPU is kept busy (not idle).
2. Throughput: number of jobs or tasks processed per unit of time.
3. Response time: amount of time needed to process an interactive command. Typically, one is interested in the average response time over all commands.
4. Turnaround time: amount of time needed to process a given task. Includes actual execution time plus time spent waiting for resources, including the CPU.

Algorithms implemented (Thread scheduling)-

First Come First Serve- In this case, threads are serviced in the order of their arrival in ready queue. As and when a thread is blocked or suspended, it is added to ready queue. When there are no threads running, one thread is picked from the ready queue in the order of their insertion

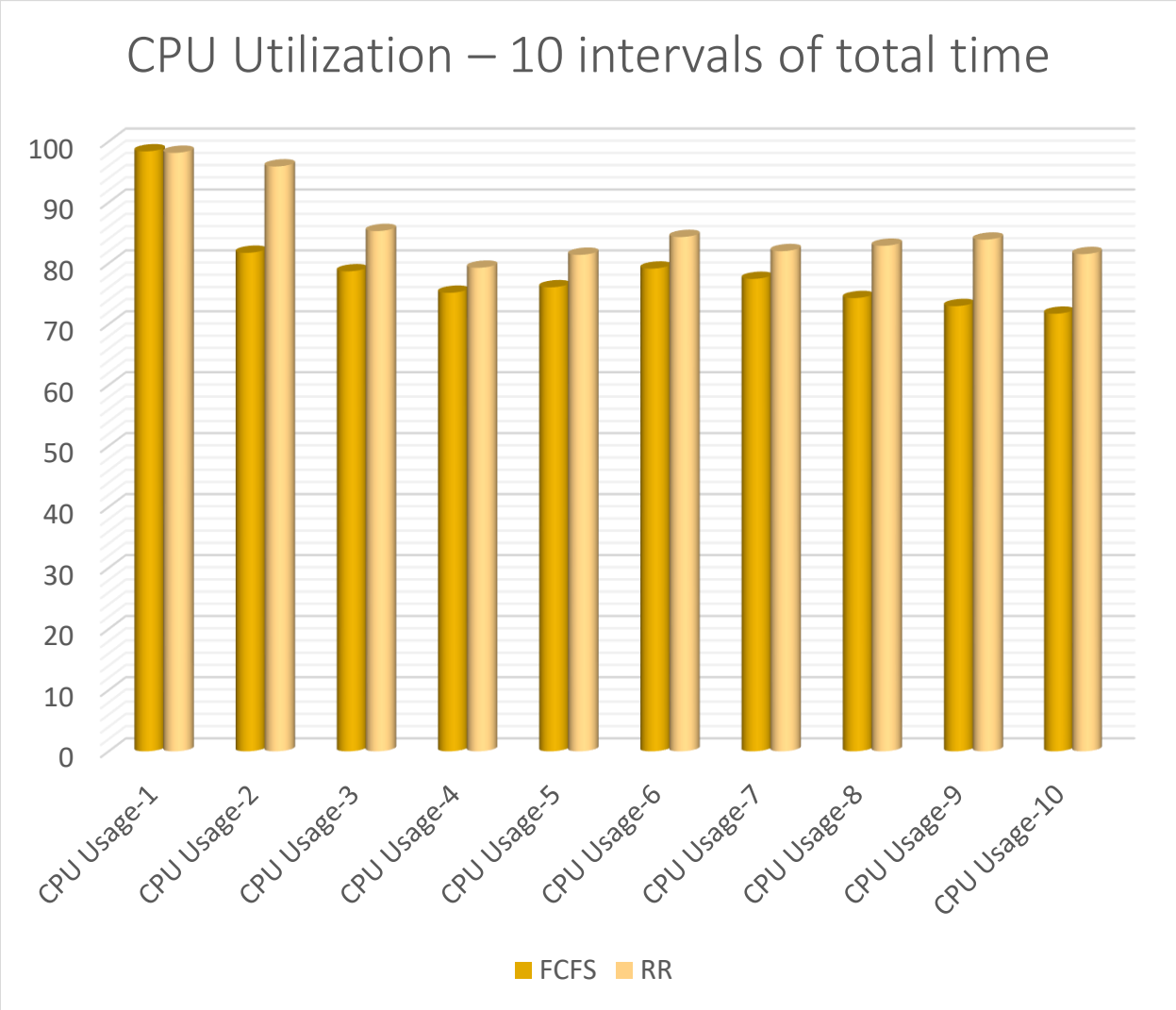
Round Robin- Here, each thread is executed for a fixed length of time known as time slice, before it is pre-empted and put back to ready queue. From the ready queue, the threads are picked in the order of arrival. The operation is similar to that of FCFS, except that each thread gets pre-empted after a fixed length of time.
In our implementation, we have set time slice to 100 clock ticks

Fixed-priority preemptive scheduling- In this case, thread with highest priority is executed among of all those threads that are currently ready to execute.
In our implementation, we have set priority randomly between 1 to 5.

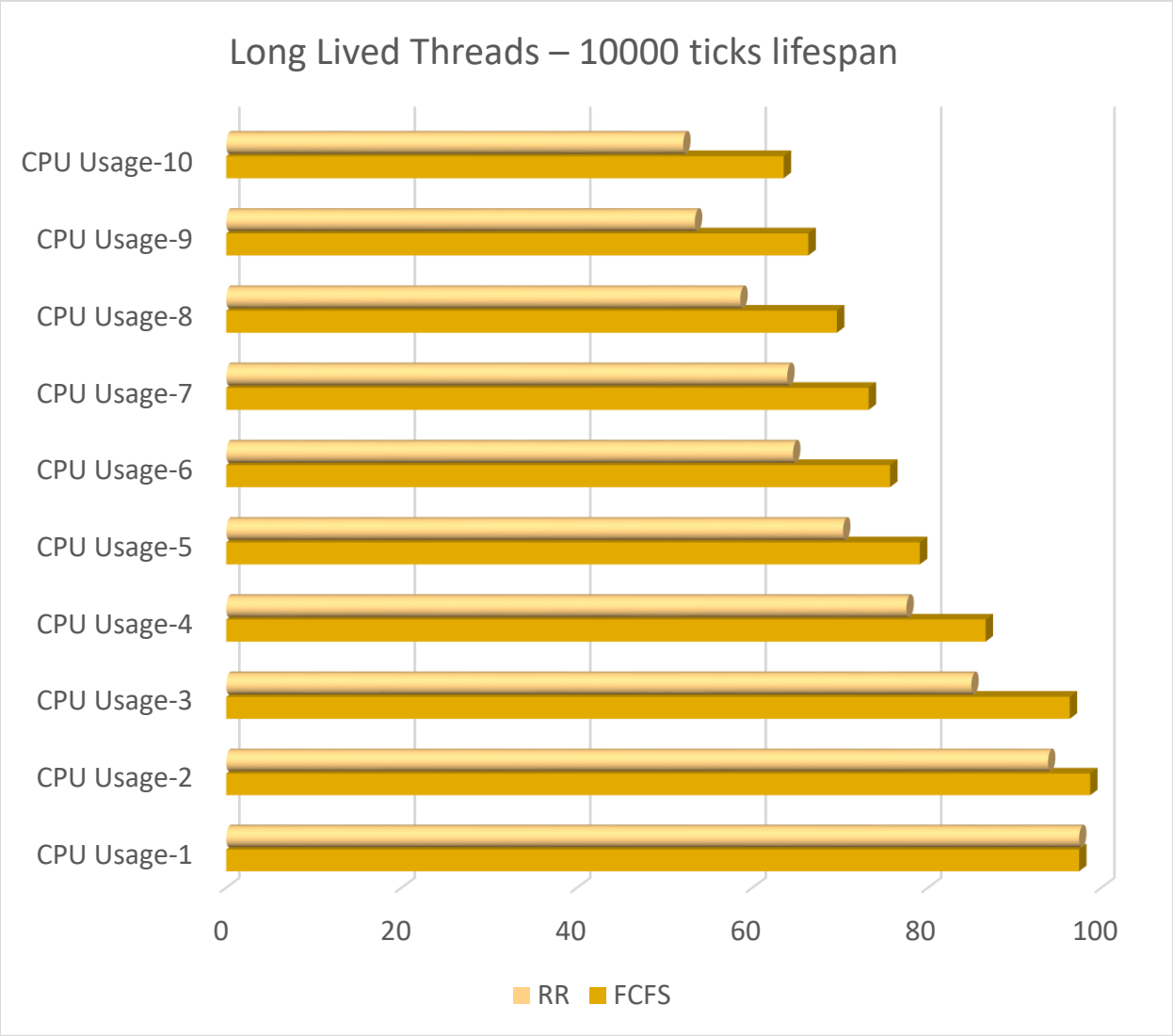
Multi-level Feedback Queue-discussed later in report

Performance Analysis-

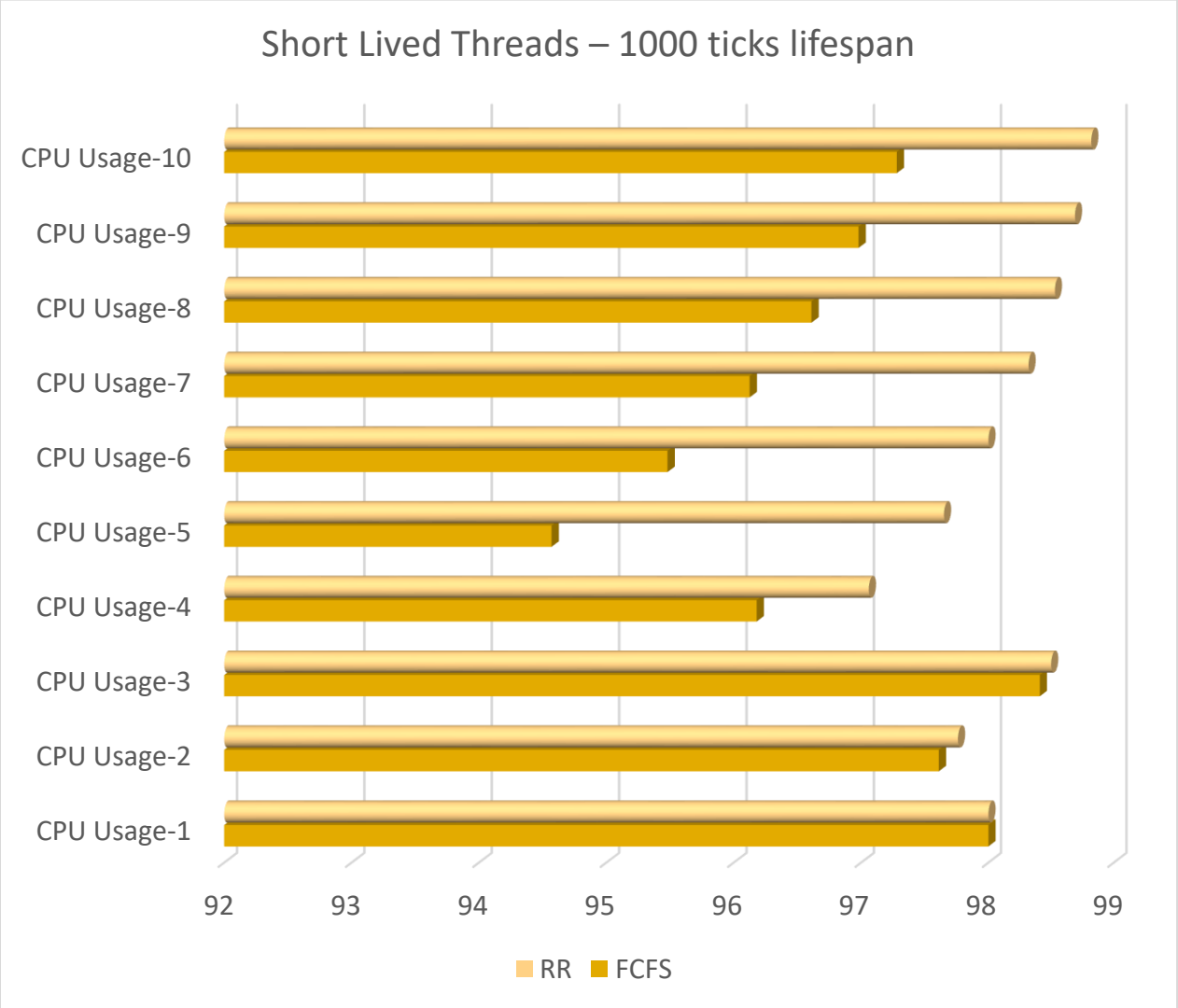
Observation 1- According to the result of simulation which is in the simulation message (saved in log file), **RR has a better CPU utilization than FCFS algorithm**.



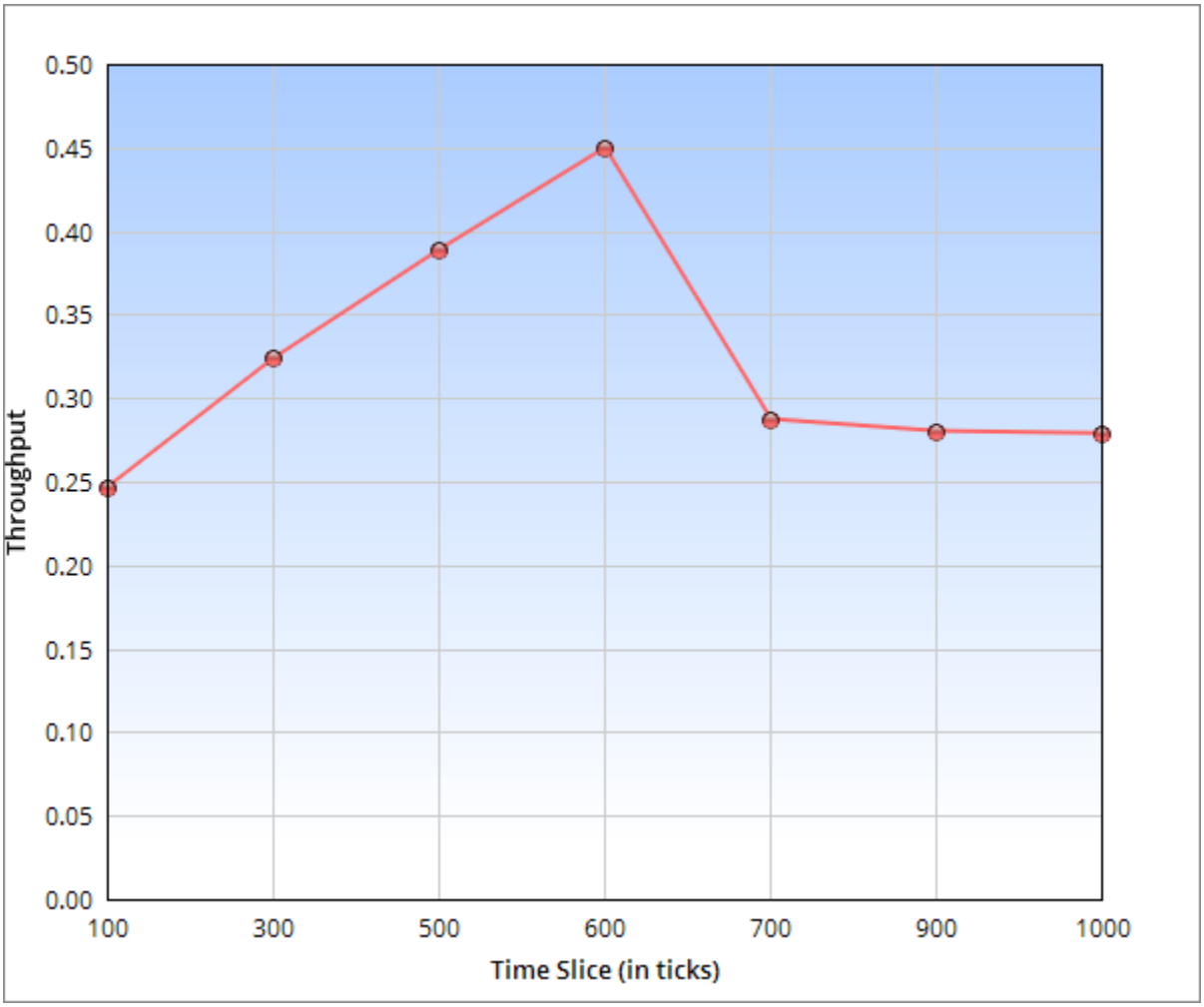
Observation 2-For long thread, RR did better than FCFS in most snapshots because it has a higher average normalized service time per thread (describes the average relative delay suffered by each thread). Because RR's time slices avoid a shorter thread to wait too long to finish, so it did better than FCFS in such a situation.



Observation 3-For short thread, FCFS did a lot better than RR because in some snapshots, it has a average normalized service time per thread far higher than RR. Since threads are all short, RR will only keep a short thread to wait longer to finish, so FCFS did better.



Observation 4: RR throughput ratio vs Time slice



Inference: Smaller the time quantum, greater is the amount of overhead required in order to switch between processes. More time we spend switching between processes, less time we spend running processes. So, **throughput is low if selected time quantum is too small**. On increasing time quantum to average needed running time, throughput increases as less no. of processes requires more than 1 RR cycle for finishing., throughput increases. **If time quantum is too long, round robin reduces to FCFS**

Optimized algorithm: Multi-level Feedback Queue (MLFQ)

We refined set of MLFQ rules as follows:

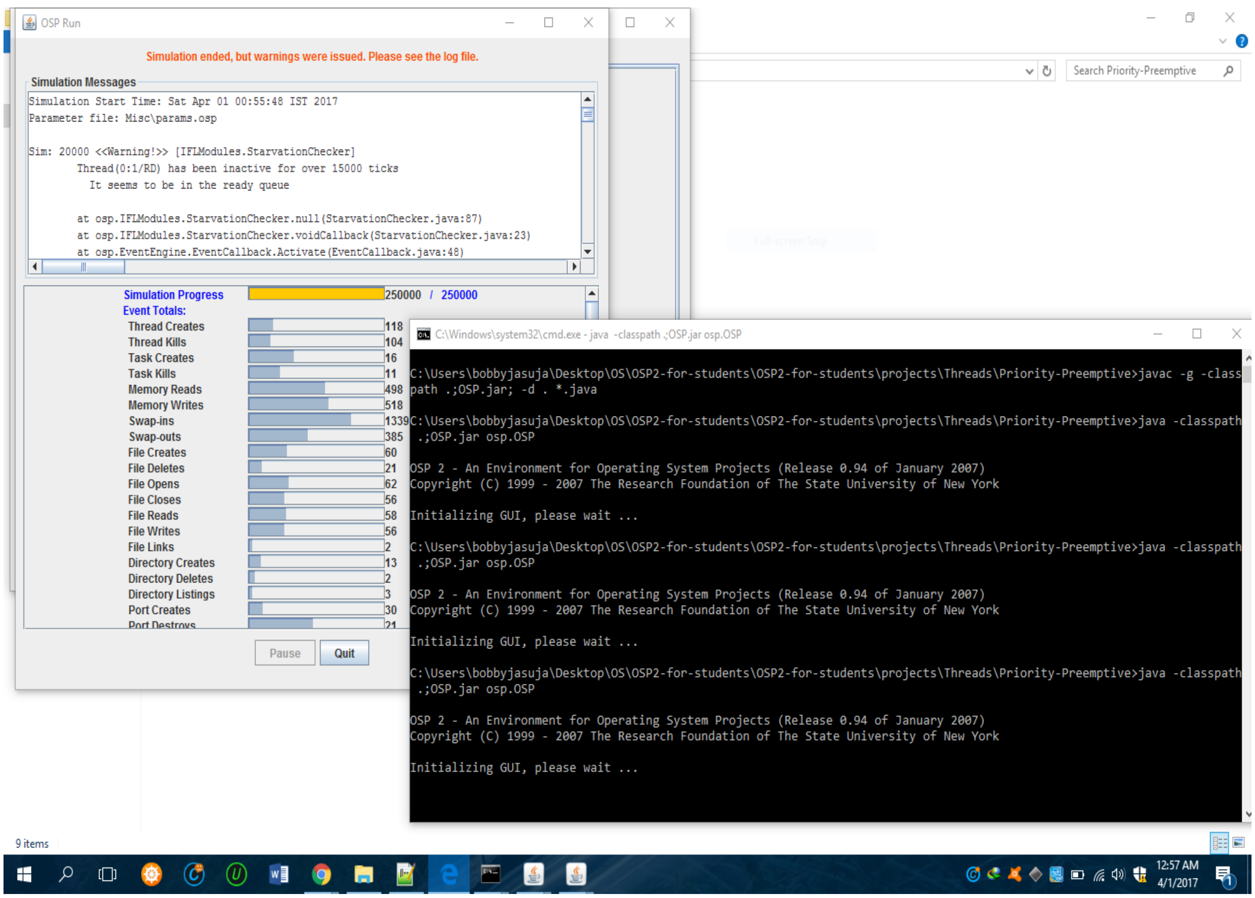
- Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
- Rule 2: If Priority(A) = Priority(B), A & B run in RR.
- Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).
- Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- Rule 5: After some time period S, move all the jobs in the system to the topmost queue.

This algorithm tries to address is two-fold problem-

a) First, we would like to optimize turnaround time.
b) Second, MLFQ would like to make a system feel responsive to interactive users (i.e., users sitting and staring at the screen, waiting for a process to finish

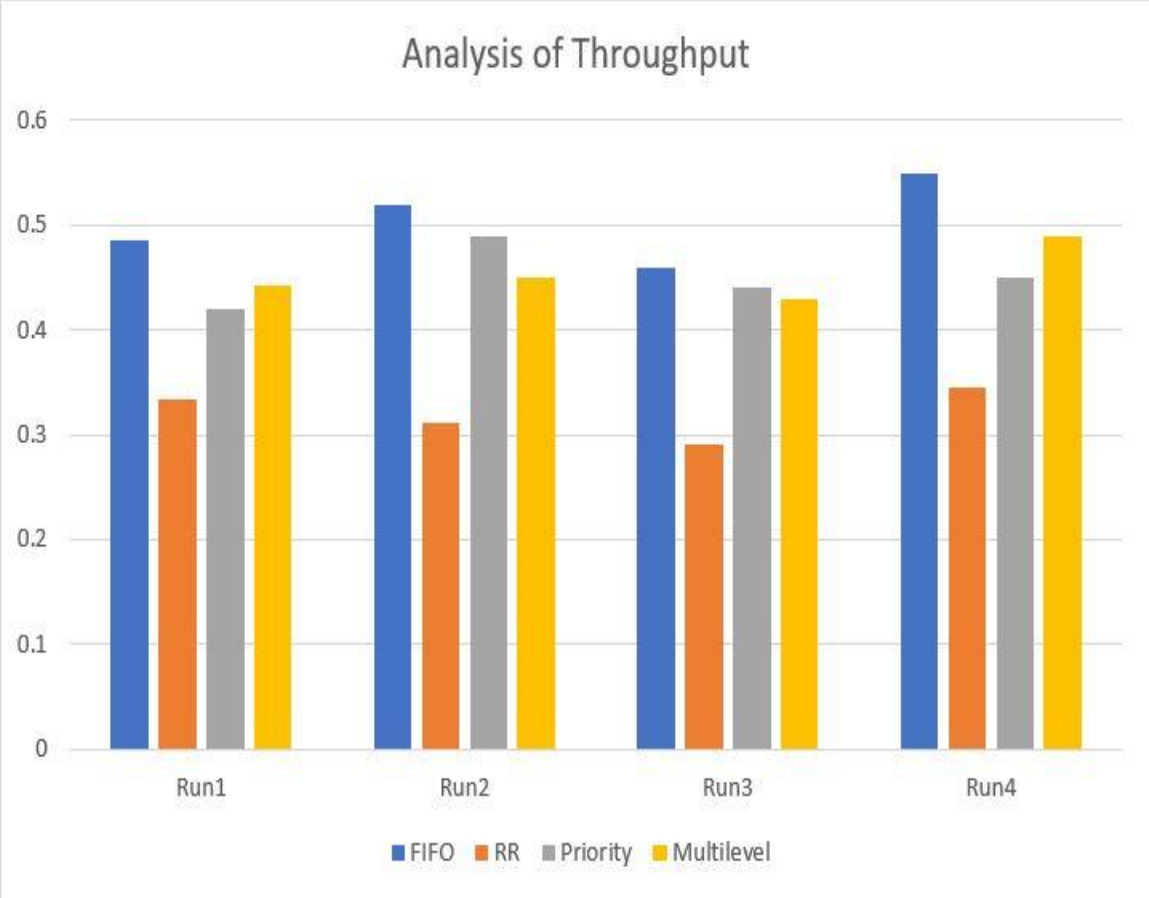
Algorithms like Round Robin reduce response time but are terrible for turnaround time whereas FCFS has good throughput but since it is non-pre-emptive algorithm response time is terrible. Priority driven has good throughput and response time but suffers from starvation.

This inference is based on values of response time, turnaround time and throughput ratio calculated in our implementation and appended in log file



Observation: On an average, FCFS algorithm gives highest throughput ratio and round robin algorithm gives lowest throughput ratio among all four whereas throughput of multi-level feedback queue and priority are in between throughputs of FCFS and round robin. Moreover, throughput ratio of priority and multi-level feedback queue are comparable to each other.

This is expected as FCFS is a non-preemptive algorithm whereas rest three are preemptive algorithms



Virtual Memory Management:

Manages physical and virtual memory using techniques such as paging and segmentation. In this module, we implemented FIFO, LRU and second chance page replacement.

Algorithm implemented (Page replacement algorithm)-
FIFO-Here, the first page which is brought into the memory is the one which is replaced.

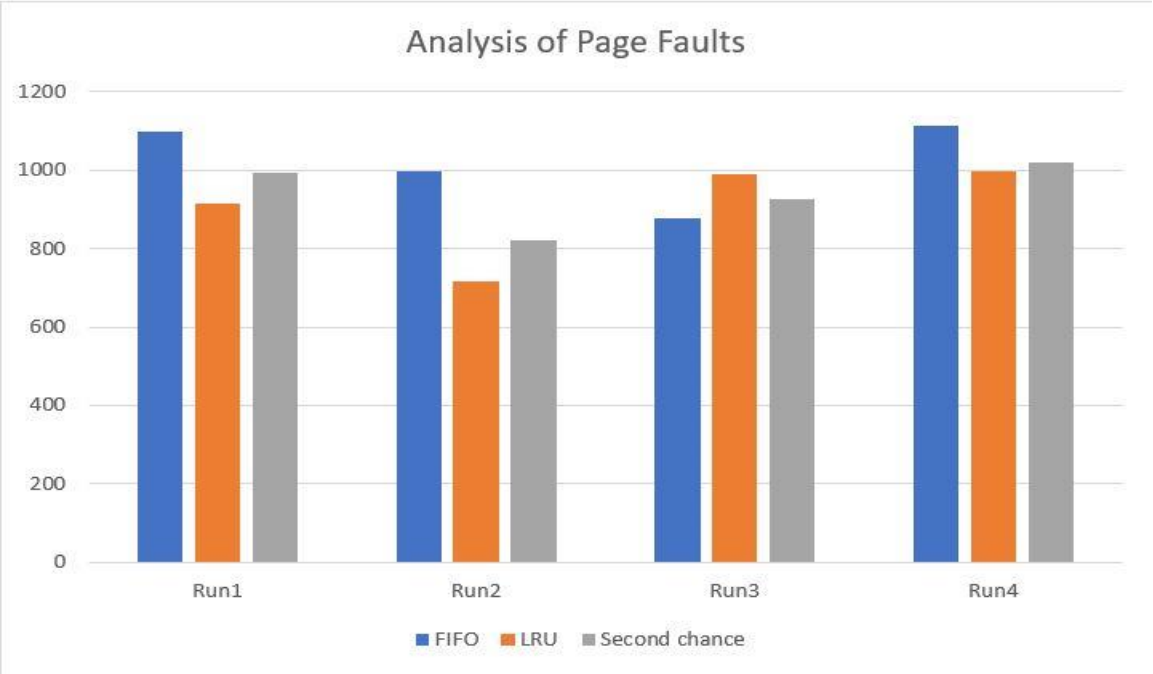
Second Chance- A modified form of the FIFO page replacement algorithm that performs better than FIFO at little cost for the improvement. It works by looking at the front of the queue as FIFO does, but instead of immediately paging out that page, it checks to see if its referenced bit is set. If it is not set, the page is swapped out. Otherwise, the referenced bit is cleared, the page is inserted at the back of the queue (as if it were a new page) and this process is repeated. This algorithm is similar to clock algorithm except that we have used array list as data structure rather than circular queue.

LRU- As the name suggests, this replacement technique is based on replacing a page which was not used recently. We maintain a past reference string and based on that we replace that has not been used for longest period of time.

Total number of page faults are appended to log file.

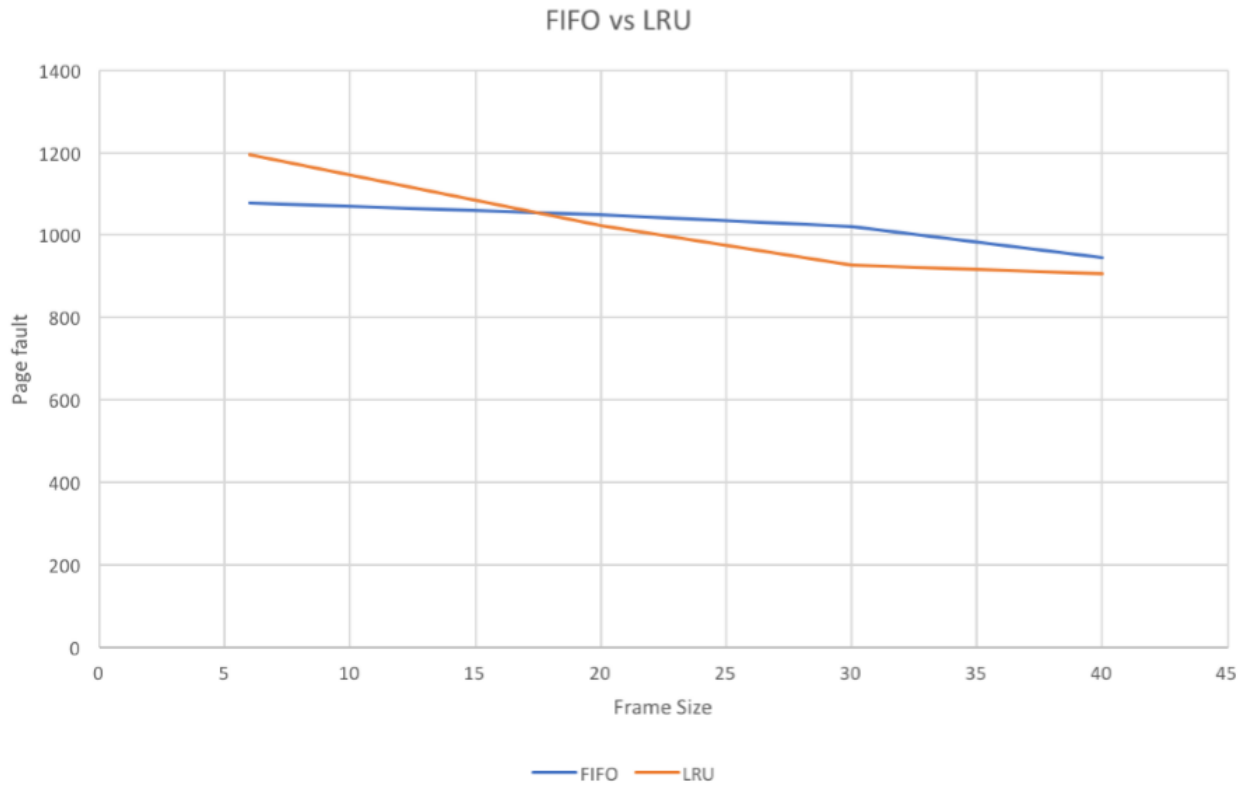
Performance analysis-

Observation 1-Tested page replacement algorithms and collect the performance data based on total no of page faults.



Inference-Overall FIFO has less overhead as there is no need to keep track of page referenced (reference string). However**, LRU turns out to be more efficient in most of real time scenarios especially in terms of total no. of page faults. Second chance being more optimized version of FIFO has total page fault count in between FIFO and LRU**

Observation 2- Page fault comparison for FIFO and LRU Algorithms when frame size is varied



Inference-LRU performs better (in terms of total no. of page faults) when the frame size increases

Scheduling of Disks (Device):

The Devices project implements certain functions of the device driver and of the basic I/O supervisor. Handles I/O requests for secondary storage devices such as disk drives.

We implemented two disk scheduling algorithms: FIFO and SSTF.

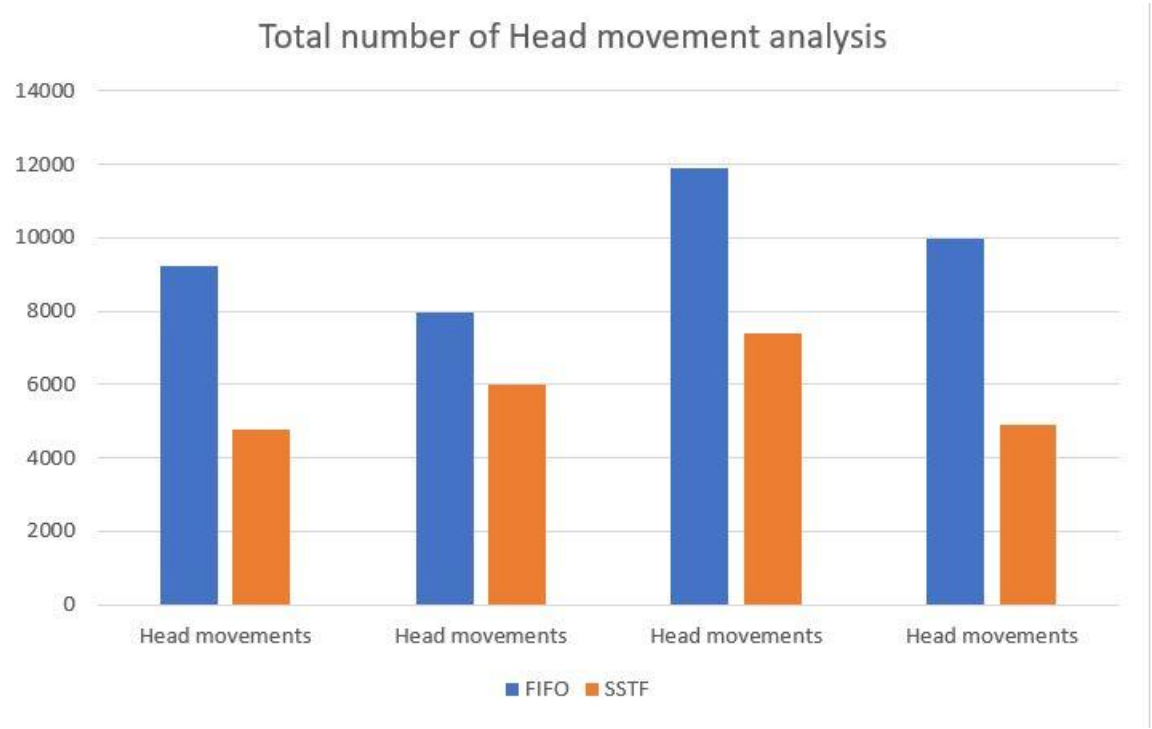
FIFO: It uses generic list as the queue structure and follows the principle of First In First Out. In this case, disk requests serviced in the order of their arrival in ready queue.

SSTF: It uses a sorted Queue (using array list) so the IORB with the least movement of the head will be dequeued first. Here the disk scheduling depends on the shortest distance from the current head position. The algorithm picks such a request where the difference between the head positions is less.

Total no. of head movements and response time of each iorb is appended in log file.

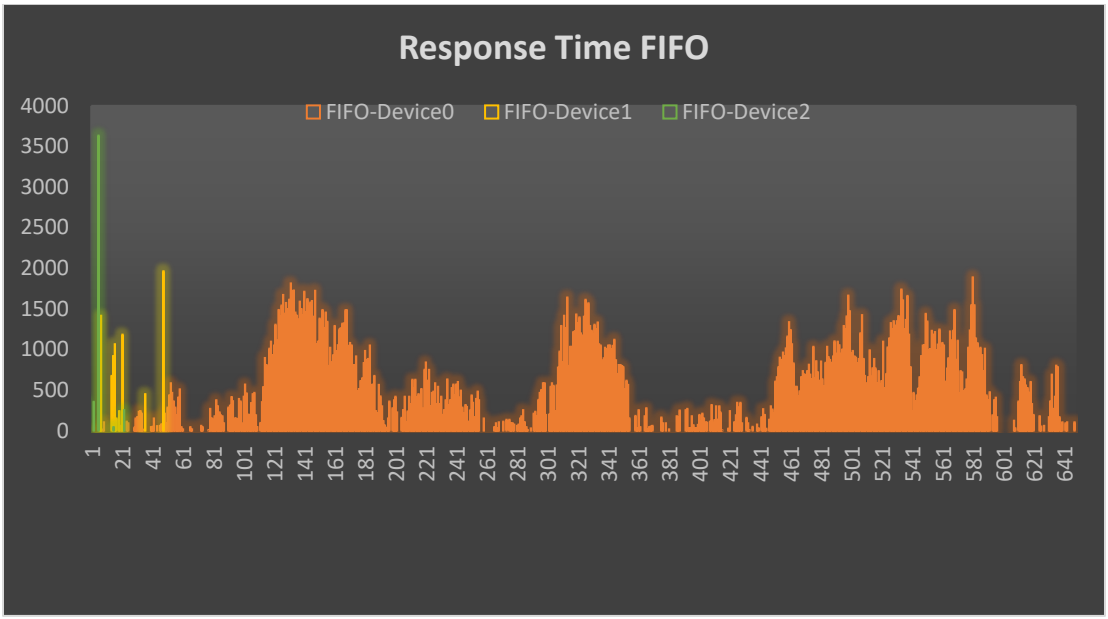
Performance Analysis:

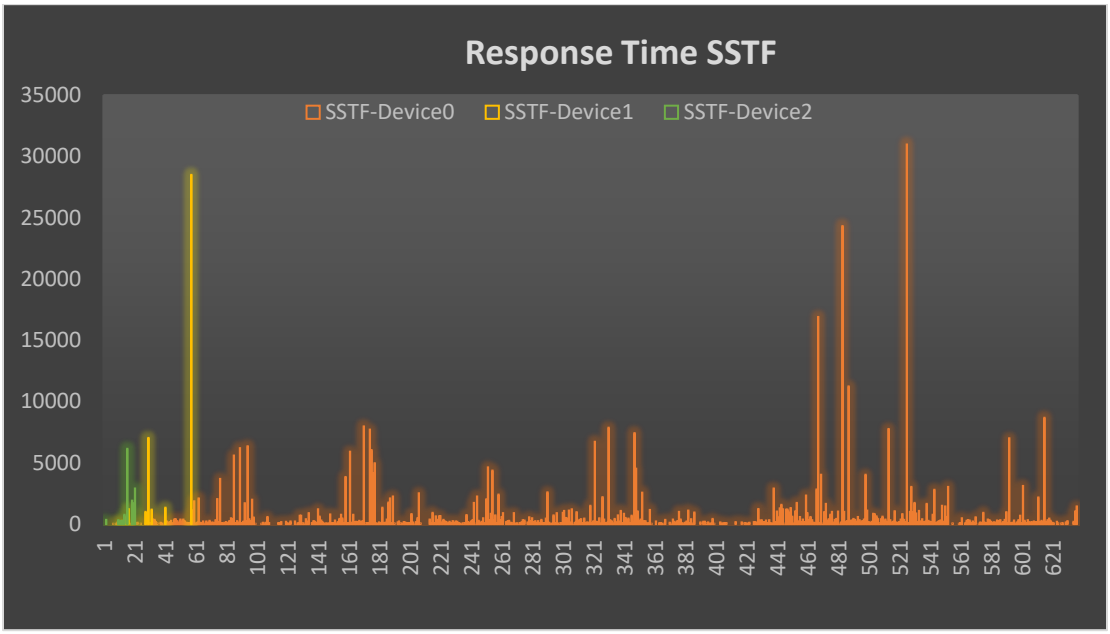
Observation 1:



Inference- SSTF performed better than FIFO (in terms of total no. of head movement)

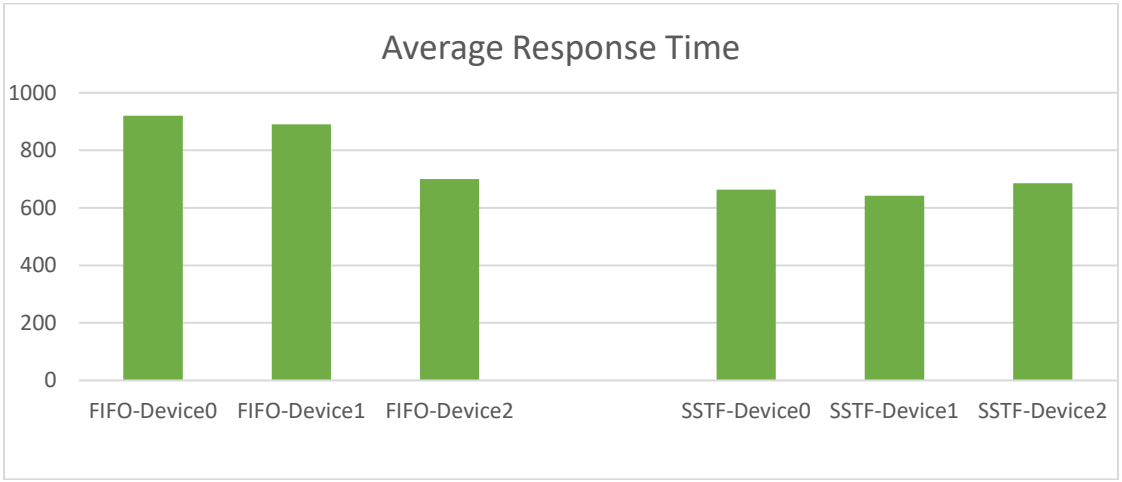
Observation 2: Collected response time of each IORB for three devices: device 0, device 1 and device 2 for each algorithm:





Inference- Most response time for SSTF are very low but at some particular points its very high because IORB with large head movement might stay in the queue for a long time

Observation 3: Average response time in of case of SSTF is lower than FIFO



File System-

This module deals with the logical layer of I/O infrastructure in an operating system. Implements the file system including basic file operations and directory structures

There are five classes in this module which we have implemented.

1. MountTable - which maps files to physical devices. We used vector to implement this
2. INode - which keeps track of space allocation to files;
3. DirectoryEntry - defines the directory structures;
4. OpenFile - provides the methods to manipulate open file handles
5. FileSys - Provides a set of operations such as create() and delete()

Snapshot-

```
FILES:
  Total number of files: 1
  Total number of directories: 3
  Total number of open files: 1
  Device 0: 1152 out of 1536 blocks are free, 25.0% used
  Device 1: 0 out of 40 blocks are free, 100.0% used
  Device 2: 0 out of 16 blocks are free, 100.0% used
```

This proves that our implementation of the file system module is correct since the simulation is successful and a list of files and other file attributes are maintained properly

Following data structure were used in this module

Class Name	Data Structures	Purpose
DirectoryEntry	Directoryentrieslist: Hash table<String, DirectoryEntry>	To access Directory entry for each inode
Inode	VectorList: Vector	To store BlockIds of the allocated blocks for a device
	BlockMappings: Boolean [] []	Block-Device Matrix to store list of Devices for each block. It stores True if Block is used and False if Block is free
	FreeBlocks: int []	Maintain a list of free blocks for a device.
	Masterlist: Genericlist	Maintains a list of inodes
OpenFile	Masterfilelist: GenericList	Maintains a list of files

Resources Mangement:

Manages abstract resources of the system using deadlock detection and deadlock avoidance algorithms.

Screenshot-

```
RESOURCES:
  Total number of acquire() calls: 145
  Total number of release() calls: 27

    Resource Availability Table
|---------------|-------------------------|
|Resource Type  |0    1    2    3    4    |
|---------------|-------------------------|
|  Total        |8    8    8    6    9    |
|---------------|-------------------------|
|  Available    |0    0    6    0    1    |
|---------------|-------------------------|


    Resource Allocation Table
|------------------|-----------------------------------|
|Resource Type     |0      1      2      3      4      |
|------------------|-----------------------------------|
|Thread ID: 109:11 |0      0      0      2      1      |
|Thread ID: 127:12 |1      3      0      0      0      |
|Thread ID: 131:12 |3      0      1      0      0      |
|Thread ID: 135:12 |1      1      0      0      2      |
|Thread ID: 136:12 |0      0      1      0      0      |
|Thread ID: 137:12 |0      0      0      3      0      |
|Thread ID: 138:11 |3      0      0      0      0      |
|Thread ID: 142:12 |0      0      0      0      5      |
|Thread ID: 149:12 |0      4      0      1      0      |
|------------------|-----------------------------------|


The suspended  RRBs:

    RRB(Id(144),Thread(148:11/W0),Resource(1),Requested(2))
    RRB(Id(140),Thread(127:12/W0),Resource(1),Requested(5))
    RRB(Id(142),Thread(146:12/W0),Resource(1),Requested(2))
```

Following functions are implemented in this module-
RRB.java-

do_grant()

In ResourceCB.java-

do_acquire()
do_deadlockDetection()
do_giveupResources()
do_release()
do_grant()

Inter-process Communication (Ports):

Implemented an inter-process communication facility that allows threads to send messages to each other

Description-

A task can create a port to serve as a mailbox. Threads from other tasks can send messages to this port. Only owner task can read from the ports of that task.

If message > PortBufferLength,

send() operation fails & message is not delivered

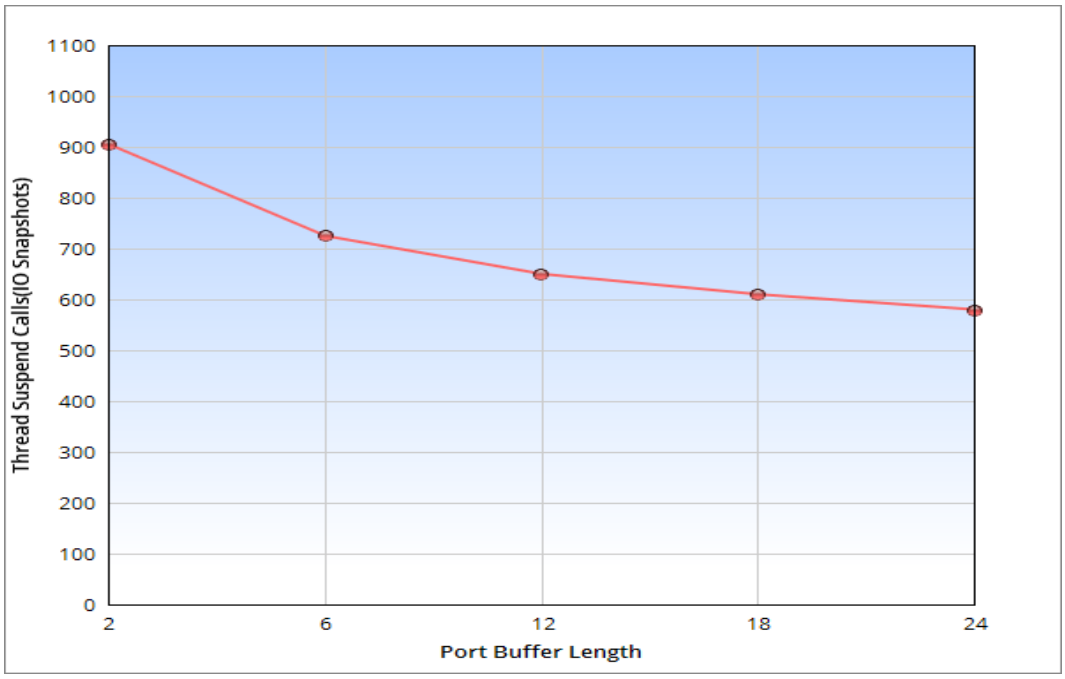
If the message < PortBufferLength,

considered well-formed & deliverable

However, the destination port might not have enough room due to other messages that might have been delivered to that port but not yet consumed. In this case, the send() operation suspends the sender thread until room becomes available.

Performance Analysis-

Observation: Thread suspend calls vs Average port buffer length



Inference: We found that thread suspend calls are decreased as port buffer length is increased.

SUMMARY

a) Management of tasks-

- CPU utilization is increased as the number of threads per task are increased
- CPU utilization is decreased as number tasks are increased
- Using different data structure did not have noticeable change in CPU Utilization

b) Management and scheduling of threads

- RR has a better CPU utilization than FCFS algorithm
- For long thread, RR did better than FCFS whereas for short thread, FCFS did better than RR
- Throughput is low if selected time quantum is too small. On increasing time quantum to average needed running time, throughput increases as less no. of processes requires more than 1 RR cycle for finishing., throughput increases. If time quantum is too long, round robin reduces to FCFS
- Algorithms like Round Robin reduce response time but are terrible for turnaround time whereas FCFS has good throughput but since it is non-pre-emptive algorithm response time is terrible. Priority driven has good throughput and response time but suffers from starvation
- On an average, FCFS algorithm gives highest throughput ratio and round robin algorithm gives lowest throughput ratio among all four thread scheduling algorithm tested

c) Virtual memory management

- LRU turns out to be more efficient in most of real time scenarios especially in terms of total no. of page faults.
- Second chance being more optimized version of FIFO has total page fault count in between FIFO and LRU
- LRU performs better (in terms of total no. of page faults) when the frame size increases

d) Implemented File system and resource management module

e) Scheduling of disks(Device)

- SSTF performed better than FIFO (in terms of total no. of head movement)
- Most response time for SSTF are very low but at some particular points its very high because IORB with large head movement might stay in the queue for a long time
- Average response time in of case of SSTF is lower than FIFO

f) Inter-process communication(Port)

- Thread suspend calls are decreased as port buffer length is increased.

BENEFITS FROM PROJECT

Operating System syllabus consists of Process Management, Memory Management, I/O Operation, File System and Scheduling. So, implementing these modules we got a better understanding on how these modules work individually and how they depend on each other. Programming skills were also tested as it was required to implement different algorithms involving different data structures in different modules.

RESOURCES USED

a) OSP 2 software.
b) Eclipse Java Neon- a Java IDE
c) Online resources for making graphs like http://www.chartgo.com

REFERENCES

a) Introduction to Operating System Design and Implementation, The OSP 2 Approach, M. Kifer, S. Smolka
b) Operating Systems: Three Easy Pieces, Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau
c) Operating Systems- Internals and Design Principles- William Stalling
d) https://en.wikipedia.org/wiki/Page_replacement_algorithm
e) https://www.youtube.com/watch?v=voiL2-nQmlU
f) http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf
g) https://megaslides.com/doc/1993223/osp2-tutorial-for-projects---southern-Illinoisuniversity