Chapter 1

Introduction

1.1 Programs and Processes

It is important to understand the concept of a process before describing schedulers. A program is a combination of instructions and data put together to perform a task when executed. Process is the program in execution. Each process has its own address space, which typically consists of program parts and data parts. The program part stores the instructions that processor executes and the data part contains data required for the process. It also includes the state of the process such as contents of the CPU registers which change dynamically with the execution of the instructions.

Life Cycle of a Process

When a process is loaded in memory, it becomes ready to execute. When the scheduler selects the process for execution, the process enters the running state. In this state, the process can either be preempted which is the case when it exceeds the time quantum allocated or blocked while waiting for I/O data. When process is preempted then the operating system puts the process on the end of the ready queue of processes, but it remains ready to execute. If the process is blocked while waiting for I/O operation, it is, then, taken from ready queue and put on the

I/O queue. When I/O channel completes the I/O operation for blocked process, the process reenters the ready state, where it waits for CPU.

Thus, at any time each process may be in one of the following states:

Ready

In this state, the process is ready to run, and waiting for CPU. This is the only state from where process can enter the running state.

Running

In this state, the process is using the CPU, and process can, either be preempted and put in the ready state, or may go to blocked state for I/O operation or may terminate with or without error.

Blocked

The process is waiting for I/O operation in this state. When channel completes its I/O operation, then, the process becomes ready and the operating system puts it on the back of the ready queue.

CPU and I/O-bound processes

A Process consists of the CPU-bound instruction and I/O-bound instructions. A process, which has the majority of the CPU-bound instructions, is called CPU-bound process. A process, which has the majority of the I/O-bound instructions, is called I/O-bound process. Hence, the success of CPU scheduling heavily depends on these characteristics of the process.

1.2 Scheduling

Scheduling is a fundamental operating system's function. CPU scheduling deals with the problem of deciding which of the process in the ready queue is to be selected for CPU. Thus, whenever the CPU becomes idle, the operating system must select from among the processes in memory that are ready to execute, and allocates the CPU to it. The part of the operating system which makes the choice as to which of the processes in the ready queue runs next is called scheduler, and the algorithm it uses is called scheduling algorithm.

At present, there are several primitive scheduling algorithms exist such as first-come first-served (FCFS), shortest job first (SJF), Priority, Round-robin, Multilevel Queue, and Multilevel Feedback Queue etc. However, there is no one best scheduling algorithm; each has its own characteristics. For example, if we can predict the next CPU burst of all the processes in the ready queue (in some way), shortest job first scheduling algorithm has minimum waiting time than others. Round-robin is basically a first-come first-served algorithm and is thus known for fairness among the processes. Here, I chose the round-robin scheduling algorithm because it is one of the most popular scheduling algorithms found in computer systems today for multiprogramming and time sharing computer environment.

1.2.1 Scheduling Goals

In terms of schedulers, there is no single definition of performance that fits everyone’s needs i.e., there is not a single performance goal for the scheduler to achieve. The many definitions of good scheduling performance often lead to a give-and-take situation, for instance, improving performance in one way decreases performance in another. Here, I am going to achieve the performance in three different ways, namely, CPU utilization, turnaround time and waiting time. CPU utilization is the percentage of time CPU is busy with processes. Turnaround time is the time difference of the arrival time and the finish time of the process. The waiting time is defined as the amount of time that a process spends waiting for CPU on the ready queue.

1.3 FCFS Scheduling Algorithm

First-come First-served scheduling algorithm is one of the simplest non-preemptive scheduling algorithms. In this algorithm, the process that requests the CPU first, gets the CPU first. The implementation of this algorithm consists of a FIFO queue of the ready processes. The process enters the ready queue and continuously moves to the front of the ready queue. When it reaches to the front of the queue, it is allocated the processor when it becomes free. This algorithm, generally, has long average waiting time. The main advantage of this algorithm is that it is easy to understand, easy to program, and ensures fairness.

1.4 Round Robin Scheduling Algorithm

It is one of the most popular scheduling algorithms found in computer systems today for multiprogramming and time sharing environment. It is similar to FCFS, but preemption is included to switch the CPU among the processes. A time duration called quantum is introduced in this algorithm, it is the time for which CPU is assigned a process. Thus, each process is assigned the same time interval (time quantum) and, if the process exceeds its time quantum, CPU is preempted and is given to another process on the ready queue.

The round-robin scheduler has the advantage of very little selection overhead as scheduling is done in constant time. Thus, scheduling decision time is simply O(1) because it has to put running process to the end of the ready queue and has to select the process from the front of the queue, which takes the constant amount of time.

What is an O(1) Algorithm?

Big-O notation is generally used to denote the growth rate of algorithms execution time based on the amount of input. For example, the running rate of an O(n) algorithm increases linearly, as the input size n grows. If it is possible to establish a constant upper bound on the running time of an algorithm, it is ,then, considered to be O(1) (constant time). That is, an O(1) algorithm is guaranteed to complete in a certain amount of time regardless of the size of the input.

The answer of the question “what makes the round-robin scheduling algorithm perform in O(1) time?” is that every time the algorithm performs exactly the same function, regardless of how many processes are on the queue. This time is referred to as the context switch time and is the time taken to save the CPU registers in the process control block for the process being preempted or blocked and restoring the CPU registers from the saved or original contents of the selected process’s control block. This allows the scheduler to efficiently select a process among many processes in the queue without increasing selection overhead cost, as the number of processes increases.

Chapter 2

Problem Statement

The performance of the Round Robin Algorithm depends heavily on the size of the time quantum. If the time quantum is large, the Round Robin simply becomes FCFS and, if time quantum is small, there are so many preemptions of the CPU. Many context switches decrease the utilization of CPU because, in case of context switch, CPU is busy with no fruitful work. Thus, we need to consider the effect of context switching on the performance of Round Robin Scheduling Algorithm.

Thus, the main purpose of my thesis is to find the optimal quantum size whereby, the utilization of the CPU is maximized and turnaround time, and waiting time for each process are minimized.

Chapter 3

Objective

The objective of my thesis work is:

To analyze the effect of quantum size on CPU utilization, turnaround time, and waiting time and, hence, to find the optimal quantum size whereby the utilization of the CPU is maximized and, turnaround time, and waiting time for each process are minimized.

Chapter 4

Methodology

4.1 Literature Survey

The evolution of scheduling closely tracked the development of computers. The concept of scheduling is not new; Henry L. Gantt, an American engineer and social scientist is credited with the development of the bar chart (Gantt Chart) in 1917 to show the performance of different scheduling algorithms.

One of the oldest, simplest, and most widely used algorithms is round-robin scheduling algorithm. There are many variations of the primitive round-robin scheduling algorithms. For example, weighted round-robin, virtual round-robin, and virtual time round-robin are the new recent versions of the round-robin scheduling algorithm.

Weighted Round Robin (WRR)

The standard round-robin does not deal with different priorities of processes. All processes are equally executed. In weighted round-robin, quantum is based on the priorities of the processes. A high prioritized process receives a larger quantum, and by this, receives execution time proportional with its priority. This is a very common extension to the primitive round-robin scheduler and will be referred to simply as the round-robin scheduler.

Virtual Round-Robin (VRR)

The virtual round-robin scheduler described by S. William [10] is an extension of the standard round-robin scheduler. The round-robin scheduler treats I/O bound processes and CPU-bound processes equally, but an I/O bound process does not fully use its time-slice and thus gets an unfair treatment compared to CPU-bound processes. The virtual round robin scheduler addresses the unfair treatment of I/O-bound processes by allowing processes to maintain their quantum when blocked, the quantum might be variable, and placing the blocked process at the front of the ready queue when it returns to the ready queue. A process is only returned to the back of the queue when it has used its full quantum. Researches have shown that this algorithm is better than the standard round-robin scheduler in terms of fairness between I/O bound processes and CPU-bound processes.

Virtual Time Round-Robin (VTRR)

The weighted round-robin and virtual round-robin schedulers both use a variable quantum for processes, as priorities are implemented by changing the quantum given to each processes. In the virtual time round-robin N. Jason [11] and T. Andrew [12] use a fixed quantum, but change the frequency by which a process is executed in order to implement priorities. This has the advantage that response times are generally improved for high prioritized processes, while the selection overhead is still constant time.

Lots of work has been done in the area of scheduler such that it should be fair among the processes according to their weights. Fairness has a meaning: given a set of jobs with associated weights, to achieve good fairness, scheduler should allocate resources to each job in proportion to its respective weight. This is reflected in work done by Larmouth [4] and [5], Newbury [6], Henry [2], and Woodside [7].

4.2 Statistics to Measure Optimality of Quantum Size

Different CPU scheduling algorithms have different properties and may favor one class of process over another. Many criteria have been suggested for comparing optimal quantum size for round robin algorithm. The criteria include the following:

i) CPU utilization

It is the percentage of time for which CPU is busy with processes. Here, we want to keep the CPU as busy as possible. Thus, if the running process requests for I/O operation, then, another process is selected to execute so that CPU is kept busy. Concept of multiprogramming is used for maximizing the CPU utilization. Several processes are kept in memory and are thus ready to run. Scheduling time is, of course, an overhead since no useful work is done. Utilization is thus measured by throughput which is measured as the number of processes completed per unit time.

ii) Turnaround Time

This is the time difference of the arrival time and the finish time of the process. It is generally the sum of the waiting time and the service time of the process. If average turnaround time decreases, then throughput will increase.

iii) Waiting Time

This is the amount of time that a process spends waiting on the ready queue. The waiting time should be kept to the minimum. Waiting time and throughput are directly dependent on each other. If average waiting time decreases, then it is clear that throughput will be increased. Average waiting time is minimal for shortest job first scheduling algorithm but, it is just like a conceptual scheduling algorithm because, we cannot find the shortest next CPU burst time of the process at run time and thus cannot implement it. We can only predict the next CPU burst time of the processes with the help of the history of that process. But this is not always accurate. Another great disadvantage of the shortest job first scheduling algorithm is that of starvation, particularly if the shortest job first (SJF) is implemented as a preemptive algorithm. In case of round robin scheduling algorithm, average waiting time will generally be not as good as in the shortest job first. In spite of that, we consider the average waiting time for round robin scheduling algorithm because it directly affects the throughput.

4.3 Algorithm Evaluation Method

There are so many scheduling algorithms, each with its own characteristics. As I have already mentioned, I used basically three criteria, namely, CPU utilization, turnaround time, and waiting time to find the optimal quantum size. Based upon these selection criteria, I used deterministic modeling. Deterministic modeling is one type of analytic evaluation method. This method takes a particular predetermined workload and defines the performance for that workload with different quantum size.

4.4 Simulator

To evaluate the round-robin scheduling algorithm, a simulator of a multiprogramming operating system (MOS) has been implemented. The goal of the MOS simulator is to make it possible to evaluate the performance of round-robin scheduling algorithm by observing the changes in the selected parameters with different quantum size. Thus, to calculate different performance parameters, we have to implement data structures for them to record the changing parameters.

The main purpose of the multiprogramming operating system (MOS) is to process a batched stream of user jobs efficiently. Another major task of the MOS is the management of hardware and software resources. These include user storage, drum storage, channel management, and the CPU. Specification and design of the MOS are given in consequent chapters.

Chapter 5

Specification

For this thesis, I have implemented a multiprogramming operating system (MOS) as a project. Specification of the MOS is based upon the ideas given by Alan C. Shaw [1]. Appendix in this text book gives the overall description of the MOS project for hypothetical computer configuration. Here, we directly deal with the basic functionalities of the operating system such as input output, interrupt handling, scheduling, main and auxiliary storage management, process and resource data structure. Description of the MOS project and its breakdown into three versions can be found in the paper by O. P. Sharma [8, 9].

5.1 Machine Specification

Every operating system provides a view of machine to its users. Similarly we can describe hypothetical configuration of the MOS computers from two points of view:

i) The virtual machine seen by the typical user.

ii) The real machine used by the MOS designer.

5.1.1 Virtual Machine

The overall configuration of the virtual machine seen by the typical user can be depicted as:

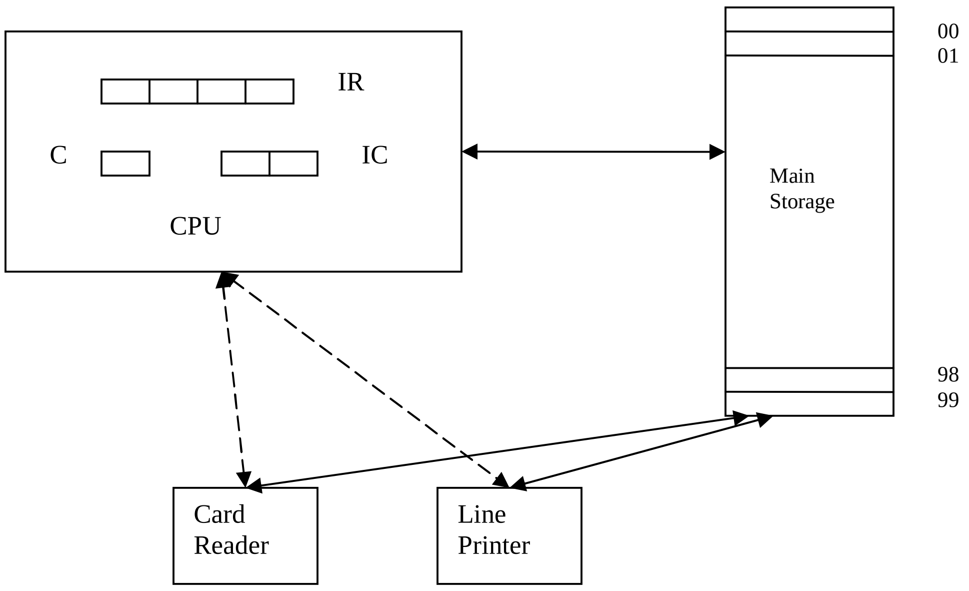

Figure 5.1: Virtual Machine

Here, we have assumed that main storage consists of maximum of hundred words, addressed from 00 to 99. Here, one word is divided into four bytes and each byte is capable of storing one character. The CPU has three registers as:

1) General purpose register which is divided into four bytes and denoted by R.
2) "Boolean" toggle having the size of one byte and denoted by C. This may contain either true "T" or false "F".
3) Instruction counter having the size of two bytes and denoted by IC, which contains the address of the next instruction to be executed.

Each instruction of the program is divided into two parts: operation code and operand address. The table below gives the format and meaning of each instruction used in our program. The first instruction of the program always begins at address 00.

Instruction		Interpretation
Operator	Operand	
LR	X_1X_2	R := [X];
SR	X_1X_2	X := R;
CR	X_1X_2	**if** R = [X] **then** C := 'T' **else** C := 'F'
BT	X_1X_2	**if** C='T' **then** IC := X
GD	X_1X_2	Read ([Z+i], i = 0,…, 9);
PD	X_1X_2	Print ([Z+i], i = 0,…, 9);
H		halt

Table 5.1: Instruction Set of Virtual Machine

Notes: 1. X_1, X_2 belongs to [0, 1, …, 9]

2. $X = 10X_1 + X_2$

3. [X] means "the contents of location X"

4. $Z = 10X_1$

We can divide these seven basic instructions into two categories: CPU-bound instructions and I/O-bound instructions. Get Data (GD) and Put Data (PD) are the examples of I/O-bound instructions whereas the remaining Load Register (LR), Store Registers (SR), Compare R (CR), Branch on True (BT), and Halt (H) are the examples of CPU-bound instructions.

Users of the machine prepare the job for batch processing by including control cards, program cards, and data cards in the sequence shown:

<JOBCard> <Program> <DATACard> <Data> <ENDJOBCard>

where <JOBCard>, <DATACard>, and <ENDJOBCard> are control cards.

<JOBCard>

<JOBCard> indicates the starting of new program which contains four entries as:

a. The $AMJ cc.1-4 A multiprogramming Job

b. <JobID> cc.5-8 a unique four character job identifier

c. <time estimate> cc 9-12, four digit maximum time estimate

d. <line estimate> cc 13-16, four digit output estimate

<Program>

Each line of the <Program> part contains information in card columns 1-40. The i^{th} card contains the initial contents of the user virtual memory locations

10(i-1), 10(i-1)+1, …….., 10(i-1)+9, i=1, 2, 3, ……., n

where n is the number of cards in the <Program> deck. The number of cards in the program deck defines the size of the user space, that is, n cards define 10*n words, n<=10. The value of n can not exceed 10 because of the size limitation of virtual memory.

<DATACard>

The <DATACard> has the format $DTA (cc. 1-4). The <DATACard> is omitted if there is no <Data> cards in the job. This control card signals end of program cards and beginning of data cards.

<Data>

The <Data> deck contains information in 1-40 and, is the user data retrieved by the virtual machine GD instruction.

<ENDJOBCard>

The <ENDJOBCard> has the format $END (cc.1-4) and <JobID> (cc. 5-8) where <JobID> should be same as in <JOBCard>. This card signifies physical end of the job deck.

5.1.2 Real Machine

The diagram of the real machine used by the MOS designer/implementer can be depicted in fig 5.2.

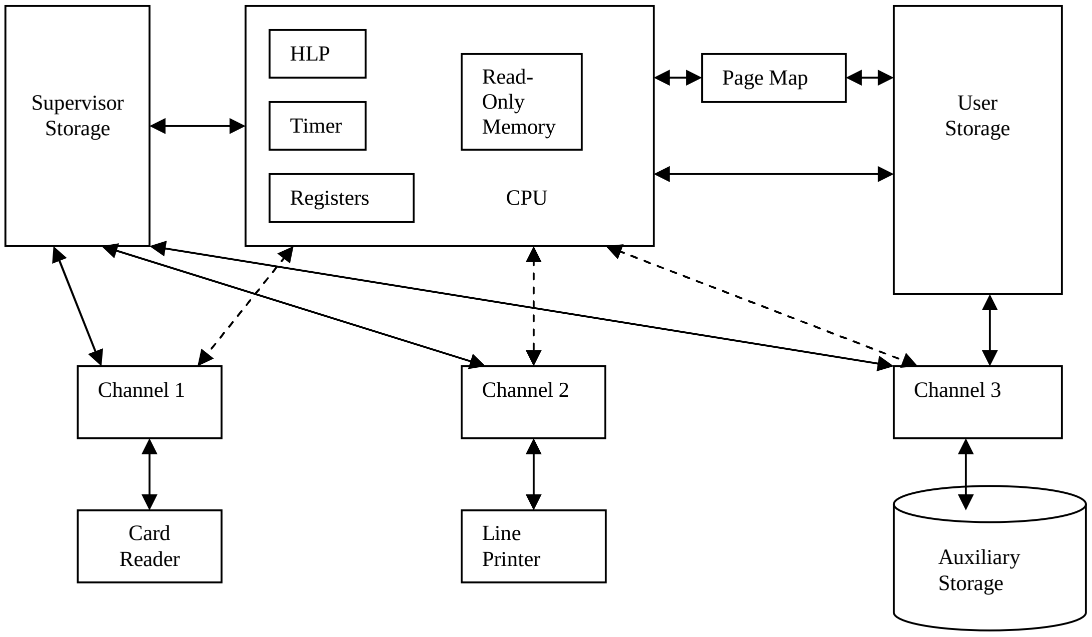

Figure 5.2: Real Machine

The overall design of the real machine can be described with the help of following subsections as:

i) Components

Figure 5.2 describes the abstract view of components of the real machine. At any time, CPU may operate in either a master mode or a slave mode. In master mode, it executes the instructions of the MOS, which resides in the supervisor storage. In slave mode, it executes the instructions of the user program which are in main memory, and accesses these programs via paging mechanism.

The CPU registers of interests are:

C: a one-byte "Boolean" toggle,

R: a four-byte general purpose register,

IC: a two-byte virtual machine location counter,

PI, SI, IOI, TI: four interrupt registers,

PTR: a four-byte page table register,

CHST[i], i=1, 2, 3: three channel status registers, and

MODE: mode of CPU, “master ” or “slave”.

Interrupt registers PI, SI, IOI, and TI are used to set the interrupts generated by user programs, channels, and timer respectively. These interrupts have been described later. PTR register is used to store the information about the page table. Channel status registers (CHST) are used to keep record of the status of the channels. At any time, channel may either be free or busy. We set the CHST[i] register to 1 if channel i is busy. The MODE register is used to store the mode of the CPU. Its value may be either 1(master mode) or 0(slave mode).

Here, main memory consists of 300 words; each word is divided into four one-byte unit. Address of each word is indicated from 000 to 299. The main memory is divided into 30 blocks where each block consists of 10 words. Supervisor storage in the Figure 5.2 indicates the amount of storage required for MOS.

The card reader and the line printer reads or writes respectively, 40 bytes of information at a time. Channel 1 and 2 are connected from peripheral devices to supervisor storage and take 5 time units to transfer information, while channel 3 is connected between auxiliary storage and both supervisor and user memory, and takes 2 time units.

The auxiliary storage is a high speed drum of 100 tracks. Each track consists of 10 words or 40 bytes. The transfer of 10 words to or from a track takes 2 time units.

ii) Master Mode Operation

Supervisor storage is used to store the main operating system. We have assumed that the master mode operations execute in zero time unit. In the master mode, interrupt registers are inspected and, the operating system accomplishes the appropriate tasks according to the value of the interrupt registers. I/O operations are initiated by starting the non busy channels with proper tasks.

iii) Slave Mode Operation

The CPU is said to be in slave mode when it is executing the user program. Each user instruction takes one time unit to execute. Paging hardware is used to map the address from virtual to real; page table is used for this purpose. The Page Table Register (PTR) points to page table location in memory and is divided into four bytes named a_0, a_1, a_2, and a_3. Here a_1 denotes the length of the page table minus one, and $10a_2+a_3$ denotes the user storage block in which the page table resides, as shown in the Figure 5.3 below.

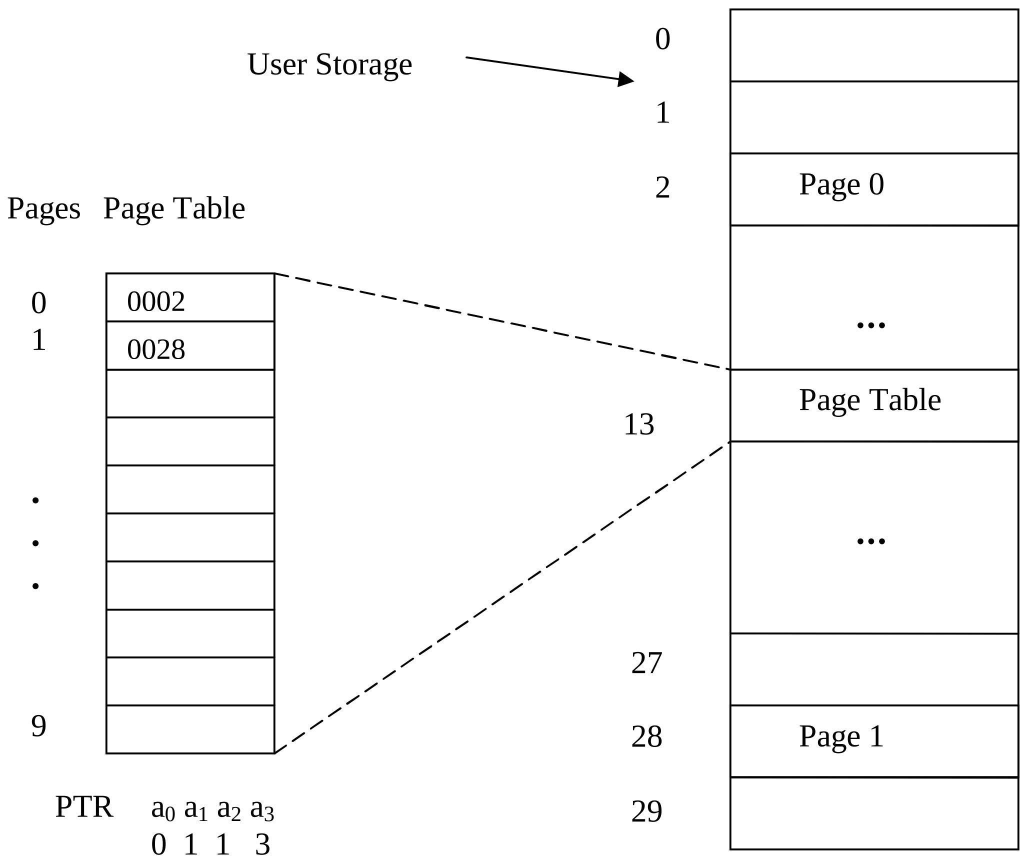

Figure 5.3: User Storage at any time

The virtual address X_1X_2 is mapped by the relocation hardware into the real address as:

$$10[10\ (10a_2 + 10a_3) + X_1] + X_2$$

iv) Channels

Channels are used for I/O operations. When MOS gives the task to the channel, the status of the channel is set to busy (1), and I/O occurs completely in parallel with CPU. After the completion of the task given to the channel by MOS, the status of the channel is reset to free, and I/O interrupt signal is raised by setting proper value in IOI registers.

Channel 1 is used to read the data from card reader into supervisor memory, channel 2 is used to print the data to the line printer form supervisor memory, and, channel 3 is used to transfer data between secondary storage and user storage as well as supervisor storage.

v) Timer

There are two time counters used in the system, namely, total time counter (TTC) and time slice counter (TSC). Total time counter is used to count the total CPU time the process has used. Time slice counter is used to count the time slice used by the running process out of the total time slice (time quantum) assigned to that process. TTC and TSC of running processes are incremented after each CPU cycle.

When the TTC of a process exceeds the total time limit of the running process as indicated by the user on the control card, the timer interrupt occurs by setting the TI register to 2. When the TSC of a particular process exceeds the time slice (time quantum) given to that process, the timer interrupt set the TI register to 1. These values are actually added to TI register. Hence TI value will vary between 0 and 3.

vi) Interrupts

Four types of interrupts can be generated.

a. **Program Interrupt (PI)**: Program interrupt, PI, is provided to indicate program errors at execution time. It occurs in slave mode.

PI=1;   interrupt due to operation code error.

PI=2;   interrupt due to operand error.

PI=3;   interrupt due to valid or invalid page fault.

b. **Supervisor interrupt (SI)**: Supervisor interrupt, SI, is provided for system calls. It occurs in slave mode.

   SI=1;   interrupt due to GD instruction.

   SI=2;   interrupt due to PD instruction.

   SI=3;   interrupt due to H instruction.

c. **Input Output interrupt (IOI)**: Input output interrupt, IOI, is provided to indicate completion of I/O operations. The different values of the IOI register when interrupt signal is raised and its interpretation are given below:

   IOI=1; when channel 1 completes its task.

   IOI=2; when channel 2 completes its task.

   IOI=3; when channel 1 and channel 2 complete their task simultaneously.

   IOI=4; when channel 3 completes its task.

   IOI=5; when channel 1 and channel 3 complete their task simultaneously.

   IOI=6; when channel 2 and channel 3 complete their task simultaneously.

   IOI=7; when all channels complete their task simultaneously.

d. **Timer Interrupt (TI)**: Timer interrupt, TI, is provided to indicate that the quantum has been finished or time limit has been finished.

   TI=1;   if quantum has been finished.

TI=2;   if time limit has been finished.

TI=3;   if both finish at the same time.

5.2 Life Cycle of a Job

In between reading the job from card reader and printing the output of the job to the line printer, the job may pass through different stages. The overall life cycle of a job can be described by three stages: input spooling, main processing, and output spooling, and is shown in figure 5.4.

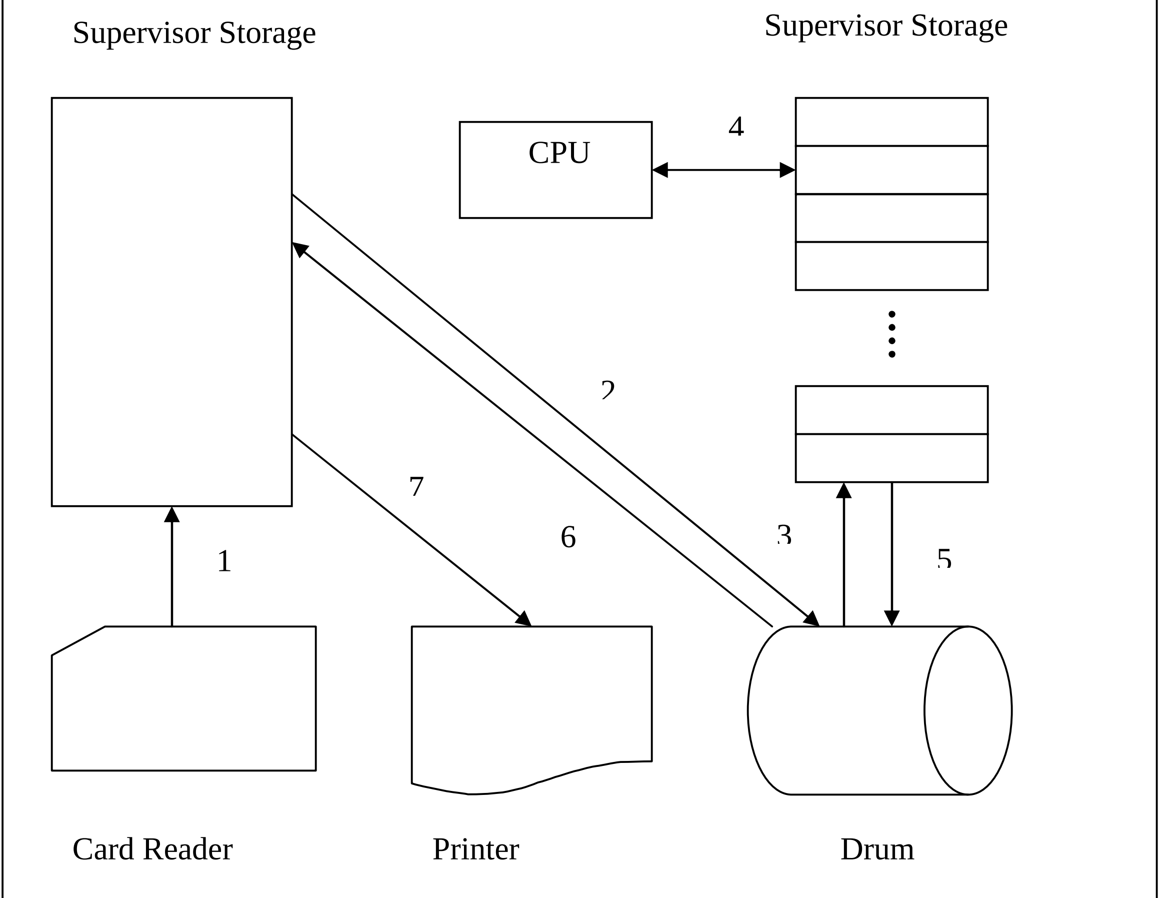

Fig. 5.4: The Life of a Job

Input Spooling

Here, program and data parts of a job are transferred from the card reader to the drum. Appropriate data structures have been maintained inside the PCB to keep record of the process and data part of the process. Channel 1 reads the input job from the card reader into the supervisor buffer and channel 3 stores these buffers into the secondary storage, drum.

Main Processing

The program part of the job is loaded from the drum track into user storage by channel 3. Then, the job is ready to run and becomes a process. During the overall life of the process while in memory, its status will generally switch many times among ready, running, and blocked. Process waits on the ready queue until scheduled, then it starts running. When GD or PD instructions execute, it is preempted and placed on blocked queue. After IO is completed by channel 3, it is moved back to ready queue. If it exceeds its time quantum, it is preempted and placed at the back of the ready queue. Finally when H is encountered or error is detected, it is moved to terminate queue.

Output Spooling

Whenever the process gets terminated, either normally or as a result of an error, outputs and error messages of the process are output spooled from drum to the printer. Channel 3 reads the output line from secondary storage into the supervisor buffer and, then, those output lines of the user program get printed by the channel 2.

Chapter 6

Design and Implementation

6.1 MOS Design

To evaluate the round-robin scheduling algorithm, a simulator of a multiprogramming operating system (MOS) has been designed. The goal of the MOS simulator is to make it possible to evaluate round-robin scheduling algorithm and the values of changing parameters with different quantum sizes.

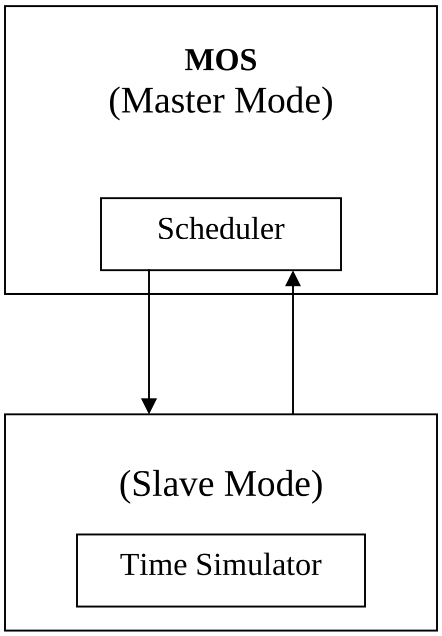

Figure 6.1: Basic Design of Multiprogramming Operating System

The focus of the MOS simulator is to test scheduler that has been implemented inside the MOS independently. An overall design of the MOS simulator is given in the Figure 6.1. There are two major modules: one is master mode and another is slave mode. The master mode handles the interrupts generated by channels, timer, and user program. When an interrupt occurs and appropriate interrupt register is set, it causes switch to master mode. In the master mode, MOS checks the value of interrupt registers and calls appropriate interrupt handling routines. Finally, after finishing the interrupt services, MOS calls the scheduler to get the ready process to run. Time simulator is used to simulate the channel and CPU timers. The overall data structures used and algorithm of main modules are given in the next subsections:

6.1.1 Data Structures used in the design of MOS

Different data structures have been conceptualized and implemented while designing and implementing the multiprogramming operating system (MOS). The main purpose of data structure is to maintain the current state of all the user processes and the current state of the operating system.

To keep record of any process, PCB has been constructed. PCB is, basically, used to keep track of all the CPU registers, time limit of the process, line limit of the process, track information of program part and data part and output messages, and outputs of the process. Additional information is also maintained here such as, CPU utilization time, waiting time, arrival time, and finish time of the process.

As the user process in multiprogramming environment goes through different states, there may be different data structures used to keep record of the processes in different states. Different queues have been maintained to keep the PCBs of the processes in different states. In case of MOS, there may be five different queues: load queue (LQ), ready queue (RQ), input output queue (IOQ), swap queue (SQ) and terminate queue (TQ). These queues can be defined as:

RQ	This is the queue which is used to store the list of ready processes. This is simply the linked list of the PCB of the different ready processes.
LQ	Whenever the process is ready to load, then it is put into the LQ. Thus LQ is a data structure which contains all the processes which are ready to load.
IOQ	Whenever the process requests for an I/O operation, it is put into the rear of the IOQ.
SQ	If process requests the frame for its further execution and if the frame is not available at that time, then it is put into the rear of the SQ.
TQ	This is the queue which is used to store the list of terminated processes but the output remains to be printed.

In case of MOS, whenever $AMJ card is read by the channel 1, operating system creates and initializes the new PCB. After reading all the program cards and data cards of the user program given and, which is indicated when $END is read by the channel 1, the operating system puts this user program on the rear of the load queue. Any process in the load queue implies that it is ready to load now. If memory frame is available and there is process in the load queue, then channel 3 simply loads the program card from the given track of the secondary storage into the indicated memory frame, and puts the PCB from load queue to the end of the ready queue. Any process in the ready queue indicates that it is ready to execute. Input output queue and swap queue are the data structures used to keep track of those process which request for the input output operations and the memory frame. All the processes, when complete their execution (either normally or abnormally due to different kinds of errors), are kept in the terminate queue. Output messages and output part of the process in terminate queue are output spooled with the help of channel 3. The PCB is deleted form the terminate queue and the process is finished if all the outputs are output spooled.

Five buffers are used which are the part of the supervisor storage. Each buffer can be used to hold up to 40 characters at any time. Initially, all buffers are placed into the empty buffer queue (EBQ). These buffers may be in one of the queues, namely, input full buffer queue (IFBQ), output full buffer queue (OFBQ). Proper data structures have been implemented to transfer buffer in between these queues.

6.1.2 MOS (Master Mode Operation)

In this case, the operating system handles the interrupts generated either in master mode or in slave mode. After handling the interrupts MOS calls for scheduler, part of the operating system, to find a new ready process to execute. Whenever scheduler gives the new ready process to the operating system, mode is switched to slave mode. Operating system runs in infinite loop. The detail operation of the operating system is given in the algorithm below:

***Algorithm**: MOS (Master Mode)*

Case TI and SI of

TI	SI	Action
0 or 1	1	Move PCB, RQ->IOQ (Read)
0 or 1	2	Move PCB, RQ->IOQ (Write)
0 or 1	3	Move PCB, RQ-> TQ
		(With error message “Normal termination”)
2	1	Move PCB, RQ->TQ
		(With error message “Time Limit Exceeded”)
2	2	Move PCB, RQ->TQ (Write)
		(With error message “Time Limit Exceeded”)
2	3	Move PCB, RQ->TQ
		(With error message “Normal Termination”)

Case TI and PI of

TI	PI	Action
0 or 1	1	Move PCB, RQ->TQ
		(With error message “Operation Code Error”)
0 or 1	2	Move PCB, RQ->TQ
		(With error message “Operand Error”)
0 or 1	3	Page Fault
		IF(Page Fault is valid and Frame available)
		Allocate the Frame

```
                              Update the Page Table
                         ELSE IF(Page Fault is valid but Frame not available)
                              Move PCB, RQ->SQ
                         ELSE (Page Fault is invalid)
                              Move PCB, RQ->TQ
                              (With error message “Invalid Page Fualt”)
2              1         Move PCB, RQ->TQ
                         (With error message
                         “Time Limit Exceeded and Operation Code Error)
2              2         Move PCB, RQ->TQ
                         (With error message
                         “Time Limit Exceeded and Operand Error)
2              3         Move PCB, RQ->TQ
                         (With error message “Time Limit Exceeded”)
```

Case TI

TI	Action
1	Move PCB to the rear of the ready queue
2	Move PCB, RQ->TQ (With error message “Time Limit Exceeded”)
3	Same as 2 above

Case IOI

IOI	Action
0	No Action
1	IR1
2	IR2
3	IR2, IR1
4	IR3
5	IR1, IR3
6	IR3, IR2
7	IR2, IR1, IR3

Finally Call **Scheduler** to select the new ready process

End Algorithm: *MOS (Master Mode*

6.1.3 Interrupt Service Routine

Interrupt service routines are the functions written for the operating system to handle the interrupts generated by the channels. In the case of MOS, there are three types of interrupt service routines IR1, IR2, and IR3 for channel 1, channel 2, and channel 3 respectively. Detailed description of these three interrupt service routines and data structures used are given below:

Algorithm*: IR1*

Read next card in given EB

```
Change status to IFB
Place on IFB (Q)
IF (not end of file and EB is available)
    Get next EB
    Start Channel 1
    Examine ifb
        IF ($AMJ)
            Create and initialize new PCB
            Allocate Frame for Page Table
            Initialize Page Table and PTR
            Set information (Program Card to follow)
            Return the IFB to EB (Q)
        IF ($DTA)
            Set information (Data Card to follow)
            Return the IFB to EB (Q)
        IF ($END)
            Place PCB on LQ
            Return the IFB to EB(Q)
        Otherwise
            Place IFB on IFB (Q)
            Save information (Program or data card of the process with JobID)
```

***END Algorithm**: IR1*

Algorithm*: IR2*

```
Print given OFB
Return OFB to EB(Q)
IF (OFB (Q) is not empty)
        Get next OFB
        Start Channel 2
```

END Algorithm*: IR2*

Algorithm*: IR3*

```
Task            Action
IS              Write given IFB into given Track
                Place track number in P or D part of PCB
                Return IFB to IB(Q)
OS              Read given Track into given EB
                Change status to OFB
                Return OFB to OFB (Q)
                Release the Track
                IF (last line)
                        Release PCB and all remaining drum tracks and memory blocks
LD              Load first program card from given track into given memory block
                Move PCB, LQ→RQ
RD              Read data card from given track into indicated memory block
```

```
                Release track
                Decrement data count in PCB
                Move PCB, IOQ→RQ
WT              Write from indicated memory block into the given track
                Increment line count in PCB
                IF( Time Exceeded)
                    Move PCB, IOQ→TQ
                ELSE
                    Move PCB, IOQ→RQ
SQ(W)           Write the victim frame into the given track
                Locate drum track with faulted page
                Task←SQ(R)
                Start Channel 3
SQ(R)           Read drum track with faulted page into the frame
                Move PCB, SQ → RQ
Now assign new task in priority order
IF ( PCB on TQ)
    IF(EB(Q) not empty)
        Get next EB from EB(Q)
    Find track number of the next output line
    Task←OS
    Start Channel 3
ELSE  IF (IFB(Q) not empty and a drum track available)
```

```
        Get next buffer from IFB(Q)
        Get a drum track
        Task←IS
        Start Channel 3
ELSE IF ( PCB on LQ and memory frame available)
        Find track number of next program card
        Allocate a frame
        Update Page Table
        Task ← LD
        Start Channel 3
ELSE IF( PCB on IOQ)
        IF(Read )
                IF(no more data card
                        Move PCB, IOQ →TQ
                        With error message "Out of data"
                ELSE
                        Find track number of next data card
                        Get memory RA
                        Task←RD
                        Start Channel 3
        ELSE IF (Write)
                IF ( line counter exceeds line limit)
                        Move PCB, IOQ→TQ
                        With error message ”Line Limit Exceeded”
                ELSE
                        Get drum track, if available
                        Update PCB
                        Find memory RA
                        Task←WR
                        Start Channel 3
ELSE IF(PCB on SQ)
        IF(memory frame available)
                Allocate
                Update Page Table
                Move PCB, SQ→RQ
        ELSE
                Run Page replacement algorithm and find a victim frame
                Allocate and Deallocate this frame by updating both page tables
                IF(victim frame not written into )
                        Locate drum track for faulted page
                        Task←SQ(R)
                        Start Channel 3
                ELSE
                        Task←SQ(R)
                        Start Channel 3
```

END Algorithm*: IR3*

6.1.4 MOS (Slave Mode Operation)

In this case, CPU is used to execute the user program and, whenever there is an interrupt in slave mode, mode is changed to master, and control is transferred to MOS. MOS saves all the current status of the process and handles the interrupts. Algorithm for slave mode operation is given below:

Algorithm*: MOS (Slave Mode Operation)*

```
LOOP
    Find the real address of IC
    IF(PI not equal to zero)
        Save the current state of process on PCB
        Give control to MOS (Master Mode Operation)
    Find the next instruction
    Increment instruction counter
    Find real address of operand of current instruction
    IF (PI not equal to zero)
        Adjust IC if necessary
        IF(PI is equal to 3 for LR and PD instruction and first time reference)
            Set "Invalid Page Fault Error" in the PCB
        Give control to MOS (Master Mode Operation)
    Case to check the operation code of the instruction
    Case        Action
    LR          R<-Memory [real address of operand of instruction]
```

```
        SR          R->Memory [real address of operand of instruction]
        CR          Compare register R and Memory [real address of operand of
                     Instruction]
                    IF (equal)      C<-True
                    ELSE            C<-False
        BT          IF (C is equal to true)
                    Set instruction counter to virtual address (operand of instruction)
        GD          Set SI equal to 1 (for input request)
        PD          Set SI equal to 2 (for output request)
        H           Set SI equal to 3 (for terminate request)
        Otherwise   Set PI equal to 1 (indicates the operation code error)
        END Case to check the operation code of the instruction
        Call for timer SIMULATION
END LOOP
```

END Algorithm*: MOS (Slave Mode Operation)*

Algorithm*: SIMULATION*

```
Increment total time counter (TTC) register
IF (Total time counter exceeds the time limit)]
        Set TI to 2
Increment time slice counter
IF (Time slice counter exceeds time quantum assigned to the process)
```

```
    Set TI to 1
FOR Channel 1
    IF (Channel 1 is busy)
        Increment Channel timer
        IF (Channel timer is equal to Channel total time)
            Set IOI as IOI+1
            (Set Channel completion interrupt)
END FOR Channel 1
FOR Channel 2
    IF (Channel 2 is busy)
        Increment Channel timer
        IF (Channel timer is equal to Channel total time)
            Set IOI as IOI+2
            (Set Channel completion interrupt)
END FOR Channel 2
FOR Channel 3
    IF (Channel 3 is busy)
        Increment Channel timer
        IF (Channel timer is equal to Channel total time)
            Set IOI as IOI+4
            (Set Channel completion interrupt)
END FOR Channel 3
IF( any of SI, PI, TI or IOI not equal zero)
```

```
    Give control to MOS (Master Mode Operation)
ELSE
    Return from SIMULATION
```

END Algorithm*: SIMULATION*

6.1.5 MOS (Scheduler)

Whenever the MOS completes all the interrupts, it calls the scheduler whose main function is to select the new ready process among the list of the ready processes. Here, in case of round robin scheduler, it takes the new process from the front of the queue. And, after assigning the CPU to the ready process, CPU is switched to the slave mode to execute that process. Before executing the process, MOS should set all the CPU registers with the help of PCB. Then, the process is allocated the CPU and, the execution begins. PCB contains all the information associated with a process.

Generally, algorithm for primitive round robin scheduler is simple because it has to select the process from the front of the ready queue. The data structure, here, for ready queue is simply the linked list of the PCBs and the variables that store the front and rear of the queue.

Algorithm*: Scheduler*

```
Select the PCB from the front of the ready queue
Maintain the ready queue
Store CPU registers in the PCB of the process which was running
```

```
Set CPU registers from the data structures maintained for PCB selected
Allocate the CPU to selected process
Switch to slave mode
```

END Algorithm: *Scheduler*

6.2 MOS Implementation

To simulate the round robin scheduler and different performance criteria, multiprogramming operating system (MOS) has been implemented as a project in C programming language. Basically, CPU control continuously switches between master mode operation and slave mode operation. MOS is interrupt driven, and when it has serviced all the interrupts, it calls the scheduler to select a user process which gets control of CPU. At the same time, mode is switched to slave. And, whenever CPU has to switch from slave mode operation to master mode operation, which occurs whenever interrupts are generated, CPU is simply preempted from the running user process and, the control is transferred to the MOS in master mode operation. PI and SI interrupt registers are set in slave mode and, by looking these values of the interrupt registers, MOS handles the interrupts. TI and IOI interrupts occur in an asynchronous fashion. Thus, CPU switches continuously between master mode and slave mode.

The CPU registers has been implemented in C programming language by declaring global variables as:

```
char R [4];       //four bytes for general purpose register
char IR[4];       //four bytes for instruction register
```

```
int  IC;           //instruction counter
char C;            //one byte for Boolean toggle
char PTR[4];       //four bytes for page table register
int SI =0;         //supervisor interrupt register
int PI =0;         //program interrupt register
int TI =0;         //timer interrupt register
int IOI=1;         //input output interrupt register
int MODE=0;        //mode of CPU: ‘slave’ or ‘master’
```

Other different parts of the MOS can be summarized as:

i) Process Control Block (PCB)

To maintain all the state of the process, structure has been implemented and all the variables are declared inside the structure for appropriate purposes. The structure for PCB node can be listed as:

```
struct PCBnode
{
     int JobID;

     char r[4];
     int  ic;
     char c;
     char ir[4];
     char ptr[4];

     int llc;
     int tll;
     int ttc;
     int ttl;

     int TrackForPage[BLOCKSIZE];
     int FaultedPage;
     int InvalidPageFault;

     struct CardListNode *PCardHead;
     struct CardListNode *PCardTail;
     struct CardListNode *PCardCurrent;

     struct CardListNode *DCardHead;
     struct CardListNode *DCardTail;
     struct CardListNode *DCardCurrent;
     struct CardListNode *OutPutCardHead;
     struct CardListNode *OutPutCardTail;
     struct CardListNode *OutPutCardCurrent;

     int ErrorMessage;

     int ArrivalTime;
     int FinishTime;
     int SetIFResponseCalculated;
     int WaitingTime;

}; //end struct PCBnode
```

Different functions have been implemented to handles the different activities of PCBs. These can be listed as:

```
     int  CreatePCB(int FrameNo);

     void InitializePCB(int pid);

     void InitializePageTableofPCB(int FrameNo);

     void InitializePTRofPCB(int pid,int FrameNo);
```

ii) Memory

User storage and auxiliary storage can be simulated in C programming language, by simply declaring global variables.

```
char M[MEMORYSIZE][WORDSIZE];

char DM[DRUMTRACKNUMBER][TRACKSIZE];
```

To keep record of either a block is free or not and either a track is free or not, block status and track status variables are declared as:

```
int  blockStatus[MEMORYSIZE/BLOCKSIZE];
int  trackStatus[DRUMTRACKNUMBER];
```

Different functions have been implemented to handle the different activities related with memories, which can be listed as:

```
     int CheckFrame(void);
```

This function is used to check for free frame in memory. If there is any free frames available in memory, then it returns TRUE, otherwise it returns FALSE.

```
     int GetFrameFromMemory(void);
```

This function is used to get frame from memory. It gives the first available free frame from use storage and makes the status of the frame allocated.

```
     int CheckTrack(void);
```

This function is used to get check for free track in memory. If there is any free tracks available in drum, then it returns TRUE, otherwise it returns FALSE.

```
     int GetTrackFromDrum(void);
```

This function is used to get track from drum. It gives the first available free track from drum and makes the status of the track allocated.

iii) Main Header File and Important Prototypes of different functions

```
#include  <stdio.h>
#include  <conio.h>
#include  <stdlib.h>
#include  <string.h>
#include  <time.h>
#include  <ctype.h>
#include  <alloc.h>

#define MEMORYSIZE       300
#define BLOCKSIZE         10
#define WORDSIZE           4
#define BUFFERNUMBER       5
```

```
#define DRUMTRACKNUMBER  100
#define TRACKSIZE         40

//used for different buffer queues
struct buffernode
{
     int BufferNo;
     struct buffernode *next;
};
//used to store information of Program card and Data card
//inside the PCB
struct CardListNode
{
     int trackNo;
     struct CardListNode *next;
};

struct ProcessTableNode
{
     int pid;
     struct ProcessTableNode *next;
     struct PCBnode *PCB;
};


struct PCBQueueNode
//PCB Queue for IS,OS,LD,IO,Swap,Terminate
{
     int pid;
     struct PCBQueueNode *next;
};


void MOS(void);
void SetRegisters(void);
void SetPCBRegisters(void);
void EXECUTEUSERPROGRAM(void);
int  ADDRESSMAP
          (struct ProcessTableNode *tempProcess,int va);


void IR1(void);
void IR2(void);
void IR3(void);

int  GetBufferFrom(int BufferQueue);
void AddBufferTo(int BufferNo,int BufferQueue);
```

```
//read next card from input file to given buffer
void ReadNextCard(int BufferNo);

struct ProcessTableNode * FindProcessTableNode(int pid);

void scheduler(void);

void StartCH1(void);
void StartCH2(void);
void StartCH3(void);

int GetPageTableLength
     (struct ProcessTableNode *tempProcess);
int GetPageTableFrameNo
     (struct ProcessTableNode *tempProcess);
void UpdatePageTableLength
     (struct ProcessTableNode *tempProcess,
          int PageTableLength);
int GetPageFrameNo
     (struct ProcessTableNode *tempProcess,int Page);
void UpdatePageFrameNo
     (struct ProcessTableNode *tempProcess,
          int Page,int FrameNo);

int FindVictimFrame(void);
void ReleasePCB(struct ProcessTableNode *tempProcess);
```

Chapter 7

Data Collection

In this section, all the data collected with the help of MOS simulator are given. The data set given by the simulator for quantum size equal to 3 is presented in this section, and all other dataset given by the simulator for quantum size equal to 1 to 5 are presented in Appendix B.

7.1 Sample Input Programs

```
$AMJ001110001000
GD40PD40LR40SR64LR41SR63LR42SR62LR43SR61
LR44SR60PD60LR40CR40BT12SR80PD80LR41SR80
PD80LR42SR80PD80LR43SR80PD80LR44SR80H
$DTA
P   I   Z   Z   A
$END0011
$AMJ001210001000
GD40LR41SR50SR51LR40SR60PD50SR61SR62LR42
SR70LR44SR71LR45SR72PD70LR42SR80LR42SR81
LR45SR82PD60PD80LR44SR90LR42PD60SR91LR43
SR92PD90PD50H
$DTA
----   | X | X   O | O
$ENDOO12
$AMJ002110001000
GD40LR40SR70PD70CR45BT00LR41SR71PD70CR45
BT00LR42SR72PD70CR45BT00LR43SR73PD70CR45
```

```
BT00LR44SR74PD70CR45BT00LR45SR75PD70CR45
BT31GD50LR50SR60LR51SR62PD60GD60PD60H
$DTA
5   4   3   2   1   0
RUN,FAST
YOU WIN
$END0021
$AMJ002210001000
GD40LR40SR70GD50LR50SR71PD70LR41SR70LR51
SR71PD70LR42SR70LR52SR71PD70LR43SR70LR53
SR71PD70LR44SR70LR54SR71PD70GD60PD60LR55
SR71PD70LR56SR62LR57SR63PD60H
$DTA
2*2=3*3=4*4=5*5=6*6=
4   9   16  25  35  36  right
This is wrong
$END0022
$AMJ003110001000
GD40LR40SR90LR41SR91PD90CR42BT38GD50LR50
SR90LR51SR91PD90CR42BT38GD60LR60SR90LR61
SR91PD90CR42BT38GD70LR70SR90LR71SR91PD90
CR42BT38GD80LR80SR90LR81SR91PD90H
$DTA
A 4 ANT END
B 4 BALL
C 4 CAT
D 4 DOG
E 4 END
$END0031
$AMJ003210001000
GD30PD30GD40LR40SR30LR41SR31LR42SR34PD30
GD50LR50SR30LR51SR31PD30H
```

```
$DTA
IF   A IS EQUAL TO B.
AND  B ITO C
THEN A I
$END0032
$AMJ004110001000
GD40GD50GD60LR40CR49BT00SR52PD50LR41CR49
BT00SR52PD50LR42CR49BT00SR52PD50LR43CR49
BT00SR52PD50LR44CR49BT00SR52PD50LR60SR52
LR61SR53LR62SR54PD50H
$DTA
0   1   2   3   4   5   6   7   8   9
This is
END bye bye
$END0041
$AMJ004210001000
GD40GD50LR50SR45LR51SR46PD40LR52SR45LR53
SR46PD40LR54SR45LR55SR46PD40LR56SR45LR57
SR46PD40LR58SR45LR59SR46PD40H
$DTA
This is your
 ha ha   hi hi   ho ho   he he   ya hoo
$END0042
```

7.2 Output of the Sample Programs

```
0011    LINE LIMIT EXCEEDED
27    PD60    P       0004
P   I   Z   Z   A
A   Z   Z   I   P
A   Z   Z   I   P
```

```
A   Z   Z   I   P

0012    NORMAL TERMINATION
40    H        X      0007
   |   |
 X | O | O
------------
 X | X | O
------------
 O | X | X
   |   |

0021    NORMAL TERMINATION
44    H       FAST    0008
5
5   4
5   4   3
5   4   3   2
5   4   3   2   1
5   4   3   2   1   0
RUN,    FAST
YOU WIN

0022    NORMAL TERMINATION
42    H       t       0008
2*2=4
3*3=9
4*4=16
5*5=25
6*6=35
This is wrong
6*6=36
```

```
This is right

0032    NORMAL TERMINATION
20    H        A I    0003
IF   A IS EQUAL TO B.
AND  B IS EQUAL TO C.
THEN A IS EQUAL TO C.

0031    NORMAL TERMINATION
46    H       END     0005
A 4 ANT
B 4 BALL
C 4 CAT
D 4 DOG
E 4 END

0042    NORMAL TERMINATION
30    H       hoo     0005
This is your        ha ha
This is your        hi hi
This is your        ho ho
This is your        he he
This is your        ya hoo

0041    NORMAL TERMINATION
39    H       bye     0006
This is 0
This is 1
This is 2
This is 3
This is 4
This is END bye bye
```

7.3 Workload and Data Set

Workload consisted of one hundred input programs made up of above eight sample programs. All individual parameters such as arrival time, finish time, and waiting time are kept inside the process control block (PCB) of the process. The workload approximately consisted of 20-30 percent I/O bound instructions and 70-80 percent CPU bound instructions.

Table: Data Set for the Workload with Quantum Size=3

PID	ARRIVAL	FINISH	TURNAROUND	WAITING
0	37	78	41	0
1	77	173	96	9
2	128	262	134	15
3	184	341	157	22
5	285	377	92	10
4	240	421	181	22
8	412	515	103	41
7	380	541	161	49
6	332	601	269	65
9	454	645	191	48
13	668	731	63	5
11	558	822	264	26
10	504	835	331	41
15	758	865	107	15
14	716	900	184	22
12	620	902	282	19
16	909	964	55	0
17	950	1078	128	12
18	999	1165	166	22
19	1049	1254	205	35
21	1149	1266	117	16
20	1112	1316	204	25
22	1203	1337	134	16
23	1293	1429	136	22
24	1329	1450	121	14
25	1368	1534	166	34
26	1416	1620	204	30
27	1472	1648	176	16

29	1571	1712	141	9
28	1529	1740	211	17
32	1741	1842	101	34
30	1657	1850	193	48
31	1707	1872	165	46
33	1781	1987	206	61
34	1831	2078	247	40
37	1983	2123	140	15
35	1885	2125	240	36
36	1943	2160	217	29
40	2193	2295	102	33
38	2117	2304	187	46
39	2157	2319	162	45
41	2231	2441	210	56
42	2281	2513	232	34
45	2432	2533	101	6
46	2552	2679	127	10
44	2392	2696	304	34
47	2570	2699	129	6
43	2334	2765	431	38
48	2604	2778	174	11
49	2708	2944	236	26
50	2758	3027	269	33
51	2809	3046	237	21
53	2907	3112	205	11
55	3067	3186	119	6
54	2983	3260	277	20
56	3115	3271	156	14
52	2867	3293	426	39
58	3209	3475	266	23
57	3153	3505	352	27
61	3386	3547	161	10
59	3288	3577	289	22
64	3600	3698	98	13
63	3566	3719	153	32
60	3348	3724	376	42
62	3514	3791	277	23
65	3638	3868	230	31
66	3719	3944	225	15
67	3769	3988	219	16
69	3891	4018	127	3
72	4027	4127	100	28
71	3997	4151	154	28
68	3847	4182	335	45
70	3957	4296	339	53
73	4069	4388	319	51
77	4329	4418	89	3

```
74   4146     4460    314         31
75   4196     4572    376         27
80   4499     4599    100         3
79   4461     4607    146         7
76   4255     4679    424         18
78   4423     4748    325         33
81   4616     4846    230         28
83   4714     4916    202         41
82   4666     4939    273         37
84   4777     5000    223         19
85   4815     5029    214         18
88   5038     5120    82          24
87   4992     5145    153         28
86   4954     5201    247         33
89   5076     5276    200         47
91   5176     5358    182         25
90   5134     5378    244         37
93   5309     5420    111         5
92   5238     5476    238         27
95   5447     5563    116         18
94   5397     5566    169         13
97   5545     5659    114         18
96   5507     5686    179         35
98   5599     5724    125         26
99   5637     5726    89          24
```

Chapter 8

Analysis

CPU utilization heavily depends upon the nature of the workload. Basically, each process in the work load might be either CPU-bound or I/O-bound. If all the processes are approximately CPU-bound, then multiprogramming environment does not help so much. Similarly CPU utilization, which usually depends upon the degree of multiprogramming, is not helped a great deal if the processes are completely or approximately 100 percent

CPU-bound. In this situation processes in the work load are approximately 70 percent CPU-bound.

FOR Quantum=1

```
Total Number of Processes--------------100

Total Context Switches-----------------3871

Total Context Switches Due to Quantum--3637

Total CPU Cycle------------------------5814

Total CPU Cycle used by the Processes--3637

Total Percentage used------------------62.555900%

Average Turnaround Time----------------201.920000

Average Waiting Time-------------------30.940000
```

Partial flow of the process with quantum size of 1 is given below. The first process PID0 enters the ready queue at time cycle 37-38 when no other process is there to compete for the processor time. Hence, PID0 is given immediate access to the processor and it starts its execution. As the quantum has been set to 1 unit, it will leave the CPU after 1 time unit but, as it is the only process in the ready queue, PID0 is again given access to the processor. In the time interval 77-78, there are now two processes in the ready queue but, being PID0 already in the ready queue, it is given access to the processor for time interval 77-78 and, PID0 is, then, put into the terminate queue. After that, PID1 is the only process until the time 128. After the time 128, both PID1 and PID2 compete for the processor time. As the time passes, the degree of multiprogramming increases and, hence, the processor utilization increases, too.

As the quantum is small, the processor is preempted from the processes very frequently. Data collected in the Chapter 7 shows that number of context switches increases as quantum size decreases. If we consider the cost of context switch, then it does decrease the percentage of CPU utilized. Waiting time and turnaround time increase in comparison with first-come first-served (FCFS), if the processes are CPU-bound.

Note: Clock t indicates the time interval t-1 to t

PID=-- indicates processor is not busy with processes

Clock=m PID=n indicates that processor is busy with the process of process id n in the time interval m-1 to m.

```
Clock PID clock PID clock PID clock PID clock PID
1     --  2     --  3     --  4     --  5     --
6     --  7     --  8     --  9     --  10    --
11    --  12    --  13    --  14    --  15    --
16    --  17    --  18    --  19    --  20    --
21    --  22    --  23    --  24    --  25    --
26    --  27    --  28    --  29    --  30    --
31    --  32    --  33    --  34    --  35    --
36    --  37    --  38    0   39    0   40    --
41    --  42    0   43    --  44    --  45    0
46    0   47    0   48    0   49    0   50    0
51    0   52    0   53    0   54    --  55    --
56    0   57    0   58    0   59    --  60    --
61    0   62    0   63    0   64    0   65    --
66    --  67    0   68    0   69    0   70    0
71    --  72    --  73    --  74    --  75    0
76    0   77    0   78    0   79    1   80    1
81    --  82    --  83    --  84    --  85    --
86    --  87    --  88    --  89    --  90    --
91    --  92    --  93    --  94    --  95    --
96    --  97    --  98    1   99    1   100   1
101   1   102   1   103   1   104   1   105   1
```

106	--	107	--	108	1	109	1	110	1
111	--	112	--	113	1	114	1	115	1
116	1	117	1	118	1	119	1	120	--
121	--	122	1	123	1	124	1	125	1
126	1	127	--	128	--	129	2	130	2
131	1	132	1	133	1	134	2	135	2
136	2	137	1	138	2	139	--	140	--
141	1	142	1	143	1	144	2	145	1
146	2	147	1	148	2	149	2	150	2
151	1	152	1	153	2	154	--	155	--
156	--	157	1	158	1	159	2	160	2
161	2	162	1	163	2	164	--	165	--
166	1	167	--	168	2	169	2	170	2
171	2	172	2	173	--	174	--	175	--
176	--	177	--	178	--	179	--	180	--
181	--	182	--	183	--	184	--	185	--
186	3	187	3	188	--	189	--	190	2
191	--	192	--	193	--	194	--	195	--
196	3	197	3	198	3	199	3	200	3
201	--	202	--	203	--	204	3	205	3
206	3	207	2	208	2	209	2	210	2
211	3	212	3	213	3	214	2	215	2
216	2	217	2	218	2	219	3	220	3
221	2	222	--	223	--	224	--	225	3
226	3	227	3	228	3	229	3	230	2
231	2	232	2	233	--	234	3	235	3

. . .

. . .

. . .

5681	96	5682	96	5683	96	5684	98	5685	99
5686	98	5687	99	5688	96	5689	96	5690	96
5691	99	5692	96	5693	99	5694	98	5695	96
5696	99	5697	98	5698	99	5699	98	5700	96
5701	99	5702	98	5703	98	5704	98	5705	98
5706	98	5707	--	5708	--	5709	--	5710	--
5711	--	5712	--	5713	--	5714	--	5715	--
5716	99	5717	99	5718	99	5719	--	5720	98
5721	98	5722	99	5723	99	5724	98	5725	98
5726	98	5727	99	5728	98	5729	99	5730	98

. . .

. . .

. . .

5781	--	5782	--	5783	--	5784	--	5785	--

```
5786  --  5787  --  5788  --  5789  --  5790  --
5791  --  5792  --  5793  --  5794  --  5795  --
5796  --  5797  --  5798  --  5799  --  5800  --
5801  --  5802  --  5803  --  5804  --  5805  --
5806  --  5807  --  5808  --  5809  --  5810  --
5811  --  5812  --  5813  --  5814  --
```

FOR Quantum=2

```
Total Number of Processes--------------100

Total Context Switches-----------------2839

Total Context Switches Due to Quantum--925

Total CPU Cycle------------------------5831

Total CPU Cycle used by the Processes--3634

Total Percentage used------------------62.322072%

Average Turnaround Time----------------201.790000

Average Waiting Time-------------------29.590000
```

As the quantum increases, almost every parameter changes as shown in the Chapter 7. Here, unnecessary context switches, due to the quantum, decrease. Thus quantum size equals to 2 definitely increases the performance than the quantum size of 1. Turnaround time and waiting time slightly decrease in this case.

FOR Quantum=3

```
Total Number of Processes--------------100

Total Context Switches-----------------2850

Total Context Switches Due to Quantum--11

Total CPU Cycle------------------------5804
Total CPU Cycle used by the Processes--3638

Total Percentage used------------------62.680910%

Average Turnaround Time----------------195.980000

Average Waiting Time-------------------25.620000
```

As we have the workload which consists of approximately 70 percent CPU-bound processes, the majority of the instructions are in the sequence of ratio 3:1 (compute vs. IO). It might be the reason why CPU utilization is maximized when the quantum equals 3. Turnaround time and waiting time heavily decreases because most processes finish their next CPU burst in a single time quantum. If the context switch is added in, the average turnaround time increases for a smaller time quantum, since more context switches are required.

FOR Quantum=4

```
Total Number of Processes--------------100

Total Context Switches-----------------2823

Total Context Switches Due to Quantum--1

Total CPU Cycle------------------------5861

Total CPU Cycle used by the Processes--3643

Total Percentage used------------------62.156629%

Average Turnaround Time----------------205.260000

Average Waiting Time-------------------27.680000
```

With quantum equals to four, unnecessary context switches heavily decrease. This shows that almost all CPU bursts are less than four. This decrement of unnecessary context switches improves the performance but, all the demerits of FCFS algorithm come up with it. That is, for the quantum size equal to four or more, our round robin algorithm behaves like FCFS scheduling algorithm.

FOR Quantum=5

```
Total Number of Processes--------------100

Total Context Switches-----------------2823

Total Context Switches Due to Quantum--0

Total CPU Cycle------------------------5861

Total CPU Cycle used by the Processes--3643

Total Percentage used------------------62.156629%

Average Turnaround Time----------------205.260000

Average Waiting Time-------------------27.680000
```

Quantum size 5 or more behaves the same as above.

Chapter 9

Graph Representation

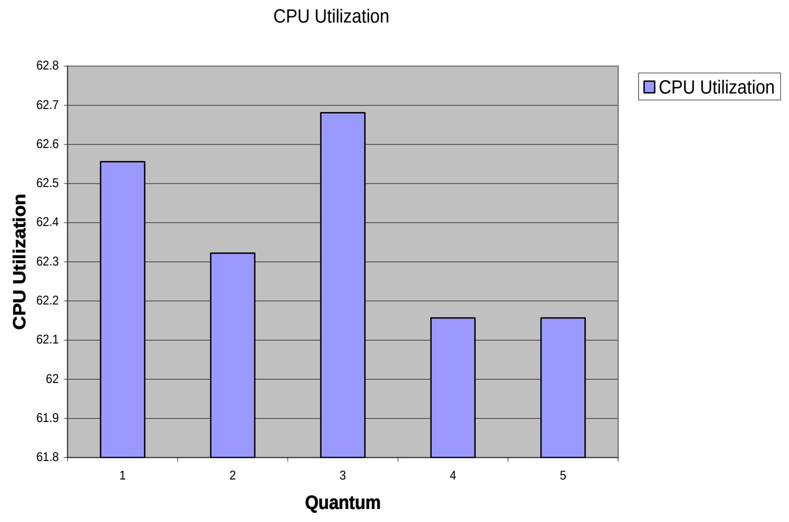

Figure 9.1: Graph relationship between CPU utilization and quantum sizes

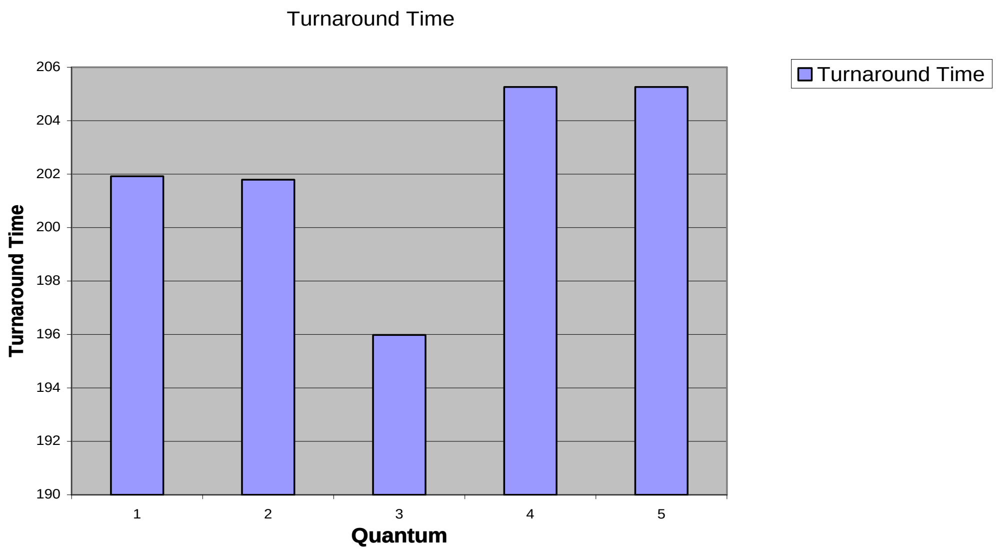

Figure 9.2: Graph relationship between turnaround time and quantum sizes

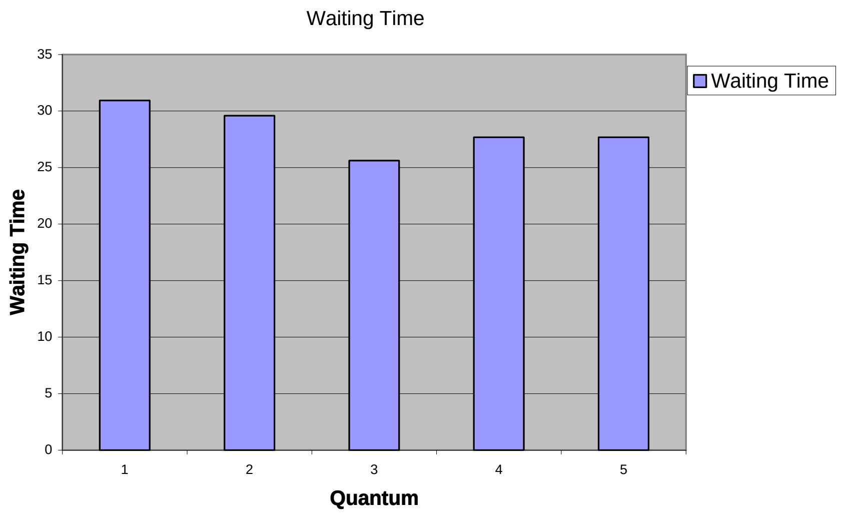

Figure 9.3: Graph relationship between waiting time and quantum sizes

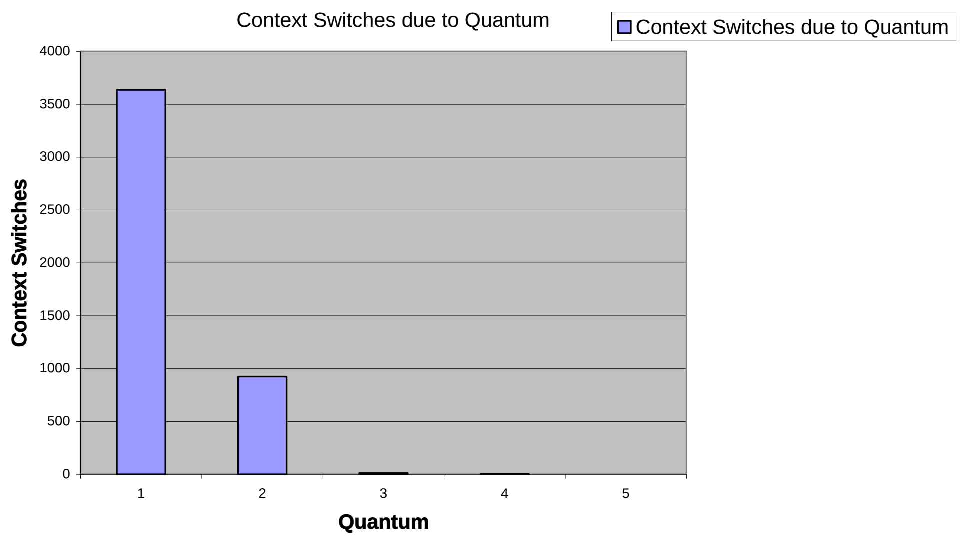

Figure 9.4: Graph relationship between context switches and quantum sizes

Chapter 10

Conclusion

CPU utilization in our simulation and workload gets maximized when the quantum size is equal to 3. Basically, our workload consists of approximately 70 percent CPU instructions and 30 percent IO instructions. Further, each sequence of CPU and I/O instructions are in ratio 3:1. Thus, almost all sequence of CPU bound instructions can execute without preemption for quantum size 3. If the quantum is set too small, then the unnecessary context switches increase heavily. And if the quantum size is large, the unnecessary context switches decrease. Thus, if we consider the effect of the context switches on the performance of the processor, the quantum size 3 worked out to be the best.

Turnaround time and waiting time heavily depend upon the nature of the processes. In case of our workload, as almost all the processes consist of approximately 70 percent CPU-bound instructions, waiting time is maximized when the quantum size is small. And, waiting time decreases when the quantum size increases. Turnaround time also increases with the quantum. The average turnaround time is improved heavily at quantum equals 3 because most processes finish their next CPU burst in a single time quantum.

Thus, if we consider the effect of all the performance measure parameters, optimal quantum size is that for which we can get the average values for all parameters. Thus, from the analysis for the workload defined earlier, we conclude that average values for all the parameters are good for quantum equal to 3, and that heavily dependent on the nature of the workload.

Recommendation

Lots of simplified assumptions have been adopted, while designing the MOS, to make the design simple. In real systems, context switch has direct impact on the performance but, here in case of MOS, it has been assumed that kernel part of the operating system runs in zero time units. This is impossible in real systems. To analyze the round robin algorithm in its entirety and to find optimal quantum size and its impact on CPU utilization, turnaround time, and waiting time, it would be better if some more realistic system were used. In case of linux operating system, so many parameters are included (inside the PCB) to keep record of almost all aspects of the process to have better statistics to measure performance. So, it might be good platform for analysis purpose of the round robin algorithm.

Fairness has become the most popular parameter for performance measure of scheduling algorithms in the recent time. Fair share scheduling has a meaning: given a set of jobs with associated weights, a fair share scheduler should allocate resources to each job in proportion to its respective weight. So many works have been done in the area of fairness. Henry [2] has contributed a lot in the field of fairness; J. Kay and P. Lauder [3], Larmouth [4] are following him and contributing in this field.

Appendix A

A.1 Source Code of the MOS in Master Mode Operation

```
void MOS(void)
{
     //MASTER:
     //CASE TI and SE of
     struct ProcessTableNode *tempProcess=NULL;
     tempProcess=FindProcessTableNode(PCBQueueRQHead->pid);

     if( (TI==0 || TI==1)  && SI==1)
     {
          SI=0;TI=0;
          //Move PCB,RQ->IOQ(Read)
          ADDpidTo(PCBQueueRQHead->pid,4);   //IOQ
          DELpidFrom(7);                     //RQ
     }
     else if( (TI==0 || TI==1)  && SI==2)
     {
          SI=0;TI=0;
          //Move PCB,RQ->IOQ(Write)
          ADDpidTo(PCBQueueRQHead->pid,4);     //IOQ
          DELpidFrom(7);                       //RQ
     }
     else if( (TI==0 || TI==1)  && SI==3)
     {
          SI=0;TI=0;
          //Move PCB,RQ->TQ(Terminate[0])
          tempProcess->PCB->ErrorMessage=0;

          ADDpidTo(PCBQueueRQHead->pid,6);      //TQ
          DELpidFrom(7);                        //RQ
     }
     else if(TI==2 && SI==1)
     {
          SI=0;TI=0;
          //Move PCB,RQ->TQ(Terminate[3]);
          tempProcess->PCB->ErrorMessage=3;

          ADDpidTo(PCBQueueRQHead->pid,6);        //TQ
          DELpidFrom(7);                          //RQ
     }
     else if(TI==2 && SI==2)
     {
          SI=0;TI=0;
          //Move PCB,RQ->IOQ(Write)then TQ(Terminate[3])
          tempProcess->PCB->ErrorMessage=3;

          ADDpidTo(PCBQueueRQHead->pid,4);         //IOQ
          DELpidFrom(7);                           //RQ
     }
     else if(TI==2 && SI==3)
     {
          SI=0;TI=0;
          //Move PCB,RQ->TQ(Terminate[3]);
          tempProcess->PCB->ErrorMessage=3;

          ADDpidTo(PCBQueueRQHead->pid,6);           //TQ
          DELpidFrom(7);                             //RQ
     }

     //CASE TI and PI of

     else if( (TI==0 || TI==1)  && PI==1)
     {
          PI=0;TI=0;
          //Move PCB,RQ->TQ(Terminate[4])
          tempProcess->PCB->ErrorMessage=4;
```

```
          ADDpidTo(PCBQueueRQHead->pid,6);      //TQ
          DELpidFrom(7);                        //RQ
     }
     else if( (TI==0 || TI==1)  && PI==2)
     {
          PI=0;TI=0;
          //Move PCB,RQ->TQ(Terminate[5])
          tempProcess->PCB->ErrorMessage=5;

          ADDpidTo(PCBQueueRQHead->pid,6);       //TQ
          DELpidFrom(7);                         //RQ
     }
     else if( (TI==0 || TI==1) && PI==3)
     {
          PI=0;TI=0;
          if(tempProcess->PCB->InvalidPageFault==0)
          {
               if( CheckFrame()==1 &&
               (tempProcess->PCB->TrackForPage
               [tempProcess->PCB->FaultedPage]==-1))
               {
                    //Allocate
                    int FrameNo=GetFrameFromMemory();
                    //Update PidForFrame[30]
                    PidForFrame[FrameNo]=tempProcess->pid;
                    //Update page table
                    int PageTableFrameNo=
                         GetPageTableFrameNo(tempProcess);
                    int PageTableLength =
                         GetPageTableLength(tempProcess);
                    UpdatePageTableLength
                         (tempProcess,PageTableLength+1);
                    int PageTablePageNo=
                         PageTableFrameNo*BLOCKSIZE+
                         (tempProcess->PCB->FaultedPage);
                    M[PageTablePageNo][0]='1';//allocate
                    M[PageTablePageNo][1]='0';///unmodified
                    //Update PageFrameNo
                    UpdatePageFrameNo
                         (tempProcess,
                         tempProcess->PCB->FaultedPage,
                         FrameNo);
                    //Adjust TrackForPage[10]
                    if(CheckTrack()==1)
                    {
                         int TrackNo=GetTrackFromDrum();
                    tempProcess->PCB->TrackForPage
                    [tempProcess->PCB->FaultedPage]=
                    TrackNo;
               }
               else
               {
                    printf("\nTracks are full");
                    exit(0);
               }
          }
          else
          {
               //Move PCB,RQ->SQ
               ADDpidTo(PCBQueueRQHead->pid,5);
               DELpidFrom(7);
          }
     }//end if(page fault valid)
     else    //if(page fault invalid)
     {
          tempProcess->PCB->ErrorMessage=6;

          ADDpidTo(PCBQueueRQHead->pid,6);//TQ
          DELpidFrom(7);//RQ
     } //end if(page fault invalid)
}
else if(TI==2 && PI==1)
{
     TI=0; PI=0;
     //Move PCB,RQ->TQ(TERMINATE(3,4);
     tempProcess->PCB->ErrorMessage=7;

     ADDpidTo(PCBQueueRQHead->pid,6);
     DELpidFrom(7);

}
else if(TI==2 && PI==2)
{
     TI=0; PI=0;
     //Move PCB,RQ->TQ(TERMINATE(3,5))
     tempProcess->PCB->ErrorMessage=8;

     ADDpidTo(PCBQueueRQHead->pid,6);
     DELpidFrom(7);
}
else if(TI==2 && PI==3)
{
     TI=0; PI=0;
```

```
          //Move PCB,RQ->TQ(TERMINATE(3))
          tempProcess->PCB->ErrorMessage=3;
          ADDpidTo(PCBQueueRQHead->pid,6);
          DELpidFrom(7);
     }
     else if(TI==1)
     {
          TI=0;
          ADDpidTo(PCBQueueRQHead->pid,7);
          DELpidFrom(7);
     }
     tempProcess=NULL;


     //CASE IOI of

     if(IOI==0)      {  /*No Action*/             }
     else if(IOI==1) { IR1();                     }
     else if(IOI==2) { IR2();                     }
     else if(IOI==3) { IR2(); IR1();              }
     else if(IOI==4) { IR3();                     }
     else if(IOI==5) { IR1(); IR3();              }
     else if(IOI==6) { IR3(); IR2();              }
     else if(IOI==7) { IR2(); IR1(); IR3();       }
     scheduler();
} //END MOS
```

A.2 Source Code for Scheduler

```
void scheduler(void)
{
   IOI=0;
   if(PCBQueueRQHead!=NULL)
   {
     struct ProcessTableNode *tempProcess=NULL;
     tempProcess=FindProcessTableNode(PCBQueueRQHead->pid);
     if(tempProcess->PCB->SetIFResponseCalculated==0)
     {
        tempProcess->PCB->ResponseTime=
          GlobalCPUTime-(tempProcess->PCB->ArrivalTime);
        TotalResponseTime=TotalResponseTime+
          tempProcess->PCB->ResponseTime;
        TotalNumberOfProcess++;
        tempProcess->PCB->SetIFResponseCalculated=1;
     }
     SetRegisters();
     EXECUTEUSERPROGRAM();
  }
}
```

A.3 Source Code of the MOS in Slave Mode Operation

```
void EXECUTEUSERPROGRAM(void)   //SLAVE MODE
{
     char operand[3]; operand[2]=NULL;

     struct ProcessTableNode *tempProcess=NULL;
     tempProcess=FindProcessTableNode(PCBQueueRQHead->pid);
     int PageTableFrameNo=GetPageTableFrameNo(tempProcess);

     char tempPTR[5];    tempPTR[4]=NULL;
     tempPTR[0]=PTR[0];  tempPTR[1]=PTR[1];
     tempPTR[2]=PTR[2];  tempPTR[3]=PTR[3];


     while(1)
     {
          RA=ADDRESSMAP(tempProcess,IC);

          if(PI!=0){   goto CHECKInterrupt;   }

          IR[0]=M[RA][0];
          IR[1]=M[RA][1];
          IR[2]=M[RA][2];
          IR[3]=M[RA][3];


          if(IR[0]!='H')
          {
               IC=IC+1;
                    operand[0]=IR[2];
                    operand[1]=IR[3];
                    operand[2]=NULL;
               if(  (!isdigit(operand[0])) ||
                    (!isdigit(operand[1])) )
               {              PI=2;                   }
               else
               {
                 RA=ADDRESSMAP(tempProcess,atoi(operand) );
               }
               if(PI!=0)
               {
```

```
          IC=IC-1;
          if(PI==3)
          {
               if((memcmp(IR,"LR",2)==0) || ]
                    (memcmp(IR,"PD",2)==0))
               {
               if(tempProcess->PCB->TrackForPage
               [(atoi(operand))/BLOCKSIZE]==-1)
               tempProcess->PCB->InvalidPageFault
                    =1;
               }
          }
          goto SIMULATION;
     }
}
if(IR[0]=='L' && IR[1]=='R')
{
     R[0]=M[RA][0]; R[1]=M[RA][1];
     R[2]=M[RA][2]; R[3]=M[RA][3];
}
else if(IR[0]=='S' && IR[1]=='R')
{
     M[RA][0]=R[0]; M[RA][1]=R[1];
     M[RA][2]=R[2]; M[RA][3]=R[3];

     M[PageTableFrameNo*BLOCKSIZE+
     (atoi(operand)/BLOCKSIZE)][1]='1';
     //set page is modified
}
else if(IR[0]=='C' && IR[1]=='R')
{
     if(R[0]==M[RA][0] && R[1]==M[RA][1] &&
     R[2]==M[RA][2] && R[3]==M[RA][3])
          C='T';
     else
          C='F';
}
else if(IR[0]=='B' && IR[1]=='T')
{
     if(C=='T')
          IC=atoi(operand);
}
else if(IR[0]=='G' && IR[1]=='D')
{
     M[PageTableFrameNo*BLOCKSIZE+
     (atoi(operand)/BLOCKSIZE)][1]='1';
     //set page is modified
```

```
          SI=1;
     }
     else if(IR[0]=='P' && IR[1]=='D')
     {
          SI=2;
     }
     else if(IR[0]=='H')
     {
          IC=IC+1;
          SI=3;
     }
     else
          PI=1;   //Operation error
//SIMULATION
     SIMULATION:
     GlobalCPUTime++;
     UtilizationCPUTime++;
     //increment the waiting time of all ready but not
     //running processes and increment total waiting
     //time
     FindWaitingTime();
     TTC++;
     if(TTC>=TTL) TI=2;
     TSC++;
     if(TSC==TS)  TI=1;

     if(CHST1==1)
     {
          CH1TimeCount++;
          if(CH1TimeCount==CH1TimeLimit)
               IOI=IOI+1;
     }
     if(CHST2==1)
     {
          CH2TimeCount++;
          if(CH2TimeCount==CH2TimeLimit)
               IOI=IOI+2;
     }
     if(CHST3==1)
     {
          CH3TimeCount++;
          if(CH3TimeCount==CH3TimeLimit)
               IOI=IOI+4;
     }
     CHECKInterrupt:

          ContextSwitch++;
```

```
                if(TI==1) ContextSwitchQuantum++;
                SetPCBRegisters();
                break;
            }
        } //end while(1)
} //END EXECUTEUSERPROGRAM
```

A.4 Source Code of the Interrupt Service Routine for Channel 1

```
void IR1(void)
{
     if(EBForCH1!=-1)
     {
          //read next card in given eb
          ReadNextCard(EBForCH1);
          //change status to ifb,place on ifb(q)
          IFBForCH1=EBForCH1;
          EBForCH1=-1;
          CHST1=0; CH1TimeCount=0;
     }
     if((!feof(INPUT_FP)) && EBQHead!=NULL)
     {
          //Get next eb
          EBForCH1=GetBufferFrom(1);//1 for EBQ
          //Start Channel 1
          StartCH1();
     }
     if(IFBForCH1 != -1)
     {
          if(memcmp("$AMJ",buffer[IFBForCH1],4)==0 )
          {
               if(CheckFrame()==1)
               {
                    //Allocate frame for Page Table
                    int FrameNo=GetFrameFromMemory();
                    //CreatePCB start
                    ISpid=CreatePCB(FrameNo);
                    //initialize PCB start
                    InitializePCB(ISpid);
                    //Initialize Page Table and PTR
                    InitializePageTable(FrameNo);
                    //Initialize PTR of the created PCB
                    InitializePTRofPCB(ISpid,FrameNo);
                    //Set F==P (Program cards to follow)
                    F=1;
                    //change status from ifb to eb and
                    //return buffer to eb(q)
                    AddBufferTo(IFBForCH1,1);
                    IFBForCH1=-1;
                    //add pid of new process to PCBQueue
                    //for IS(input spooling
               }//end if(CheckFrame()==1)
               else //frame not available
               {
                    if(EBForCH1 != -1)
                    {
                         CHST1=0;
                         AddBufferTo(EBForCH1,1);
                         EBForCH1=-1;
                    }
               }//end frame not available
          }
          else if(memcmp("$DTA",buffer[IFBForCH1],4)==0)
          {
               //setF<--D(data cards to follo)
               F=2;
               //change status from ifb to eb and return
               //buffer to eb(q)
               AddBufferTo(IFBForCH1,1);
               IFBForCH1=-1;
          }
          else if(memcmp("$END",buffer[IFBForCH1],4)==0)
          {
               //Place PCB on LQ,
               ADDpidTo(ISpid,3);//3 for LD queue
               //change status from ifb to eb and return
               //buffer to eb(q)
               AddBufferTo(IFBForCH1,1);
               IFBForCH1=-1;
          }
          else
          {
               //place ifb on ifb(q)
               AddBufferTo(IFBForCH1,2);
               IFBForCH1=-1;
               //save F information(program or data card
               //for CH3)
               AddCardTypeToFQueue(F);
               AddPidToPQueue(ISpid);
          }
     }
}
```

A.5 Source Code of the Interrupt Service Routine for Channel 2

```
void IR2(void)
{
     if(OFBForCH2!=-1)
     {
          //print given ofb
          for(int i=0;i<40;i++)
          {
               fputc(buffer[OFBForCH2][i],OUTPUT_FP);
          }
          fputc('\n',OUTPUT_FP);
          AddBufferTo(OFBForCH2,1);
          OFBForCH2=-1;
          CHST2=0;
          CH2TimeCount=0;
     }
     if(OFBHead!=NULL)
     {
          OFBForCH2=GetBufferFrom(3);
          StartCH2();
     }

     else if(  PCBQueueISHead==NULL &&
               PCBQueueLDHead==NULL &&
               PCBQueueIOHead==NULL &&
               PCBQueueSQHead==NULL &&
               PCBQueueTQHead==NULL &&
               PCBQueueRQHead==NULL &&
               CHST1==0 && CHST3==0 &&
               CHST2==0 && OFBHead==NULL )
               {

                    exit(0);
               }
}
```

A.6 Source Code of the Interrupt Service Routine for Channel 3

```
void IR3(void)
{
     if(Task!=0)
     {
          CHST3=0;
```

```
CH3TimeCount=0;
struct ProcessTableNode *tempProcess=NULL;
if(Task==1)         //IS
{
     tempProcess=
          FindProcessTableNode(PQueueHead->pid);
     Strncpy
          (DM[TrackNoForIS],buffer[EBForIS],40);
     PlaceTrackNoToPorD
          (tempProcess,TrackNoForIS);
     AddBufferTo(EBForIS,1);//1 for EBQ
     EBForIS=-1;  TrackNoForIS=-1;
}
else if(Task==2)//OS
{
     tempProcess=
     FindProcessTableNode(PCBQueueTQHead->pid);
     if(ErrorCount==0)
     {
          FindErrorMessage(tempProcess,EBForOS);
          ErrorCount++;

          AddBufferTo(EBForOS,3);
          EBForOS=-1;
     }
     else if(ErrorCount==1)
     {
          FindErrorMessage(tempProcess,EBForOS);
          ErrorCount++;

          AddBufferTo(EBForOS,3);
          EBForOS=-1;
     }
     else
     {
          strncpy
          (buffer[EBForOS],DM[TrackNoForOS],40);
          AddBufferTo(EBForOS,3);   //3 for OFB
          //Release Track
          trackStatus[TrackNoForOS]=0;

          tempProcess->PCB->OutPutCardCurrent=
               tempProcess->PCB->
               OutPutCardCurrent->next;
          if(tempProcess->PCB->
               OutPutCardCurrent==NULL)
          {
```

```
                    ErrorCount=0;
                    ReleasePCB(tempProcess);
                    DELpidFrom(6);
               }
               EBForOS=-1; TrackNoForOS=-1;
          }//else if ErrorCount!=0 or ErrorCount!=1
     }
     else if(Task==3)//LD
     {
          tempProcess=
          FindProcessTableNode(PCBQueueLDHead->pid);
          Strncpy
               (M[FrameNoForLD*BLOCKSIZE],
               DM[TrackNoForLD],40);

          //Update PidForFrame[30]
          PidForFrame[FrameNoForLD]=tempProcess->pid;

          //update the arrival time of the process
          tempProcess->PCB->ArrivalTime=GlobalCPUTime;

          ADDpidTo(tempProcess->pid,7);
          DELpidFrom(3);

          FrameNoForLD=-1; TrackNoForLD=-1;
     }
     else if(Task==4)//RD
     {
          tempProcess=
          FindProcessTableNode(PCBQueueIOHead->pid);
          strncpy(M[RAForIO],DM[TrackNoForIO],40);
          //Release Track
          trackStatus[TrackNoForIO]=0;
          tempProcess->PCB->DCardCurrent=
               tempProcess->PCB->DCardCurrent->next;
          ADDpidTo(tempProcess->pid,7);
          DELpidFrom(4);
          RAForIO=-1;  TrackNoForIO=-1;
     }
     else if(Task==5)//WT
     {
          tempProcess=
          FindProcessTableNode(PCBQueueIOHead->pid);
          strncpy(DM[TrackNoForIO],M[RAForIO],40);
          tempProcess->PCB->llc++;
          if(tempProcess->PCB->ttc>=
               tempProcess->PCB->ttl)
          {
               ADDpidTo(tempProcess->pid,6);    //TQ
               DELpidFrom(4);                   //IO
          }
          else
          {
               ADDpidTo(tempProcess->pid,7);     //RQ
               DELpidFrom(4);                    //IO
          }
          RAForIO=-1; TrackNoForIO=-1;
     }
     else if(Task==6)//SQ(W)
     {
          strncpy
          (DM[TrackNoForSQW],M[VFForSQW*BLOCKSIZE],
          40);
          FForSQR=VFForSQW;
          TrackNoForSQW=-1;  VFForSQW=-1;
          Task=7;
          StartCH3();
          return;
     }
     else if(Task==7)              //SQ(R)
     {
          tempProcess=
          FindProcessTableNode(PCBQueueSQHead->pid);
          TrackNoForSQR=tempProcess->PCB->TrackForPage
               [tempProcess->PCB->FaultedPage];
          strncpy
          (M[FForSQR*BLOCKSIZE],DM[TrackNoForSQR],40);
          //Move PCB,SQ->RQ after setting TSC<-0
          ADDpidTo(tempProcess->pid,7);      //RQ
          DELpidFrom(5);                     //SQ
          TrackNoForSQR=-1;  FForSQR=-1;
     }
     Task=0;//reset the task
}//end if(Task!=0)

struct ProcessTableNode *tempProcess=NULL;
//(Now Assign New Task in Priority Order)

if(PCBQueueTQHead!=NULL  && EBQHead!=NULL)
//(output spool first)
{
     tempProcess=NULL;
     tempProcess=
          FindProcessTableNode(PCBQueueTQHead->pid);
     EBForOS=GetBufferFrom(1);
     if(ErrorCount==0 || ErrorCount==1)
     {
          Task=2;    StartCH3();  return;
     }
     if(tempProcess->PCB->OutPutCardCurrent==NULL)
     {
          ErrorCount=0;
          AddBufferTo(EBForOS,1);
          ReleasePCB(tempProcess);
          DELpidFrom(6);//TQ
          return;
     }
     TrackNoForOS=
     tempProcess->PCB->OutPutCardCurrent->trackNo;
     Task=2;//2 for OS
     StartCH3();
}
else if(IFBHead!=NULL && CheckTrack()==1)
{
     //Get next buffer from ifb(q)
     EBForIS=GetBufferFrom(2);
     //Get drum track
     TrackNoForIS=GetTrackFromDrum();
     Task=1;
     StartCH3();
}
else if(PCBQueueLDHead!=NULL  && CheckFrame()==1)
{
     tempProcess=NULL;
     tempProcess=
          FindProcessTableNode(PCBQueueLDHead->pid);
     //Find track number of next program card
     TrackNoForLD=
          tempProcess->PCB->PCardCurrent->trackNo;
     //Allocate a frame
     FrameNoForLD=GetFrameFromMemory();
     //Update Page Table
     int PageTableLength;
     if(tempProcess->PCB->ptr[1]==NULL)
          PageTableLength=0;
     else
     {
          PageTableLength=
               GetPageTableLength(tempProcess);
          PageTableLength=PageTableLength+1;
     }
     UpdatePageTableLength
          (tempProcess,PageTableLength);
     UpdatePageFrameNo
          (tempProcess,PageTableLength,FrameNoForLD);
     Task=3;
     StartCH3();
}
else if(PCBQueueIOHead!=NULL)
{
     tempProcess=
          FindProcessTableNode(PCBQueueIOHead->pid);
     if(  tempProcess->PCB->ir[0]=='G' &&
          tempProcess->PCB->ir[1]=='D')
     {
          if(tempProcess->PCB->DCardCurrent==NULL)
          {
               //out of data message
               tempProcess->PCB->ErrorMessage=1;
               //6 for TQ
               ADDpidTo(tempProcess->pid,6);
               DELpidFrom(4);//4 for IO
          }
          else
          {
               TrackNoForIO=
               tempProcess->PCB->
                         DCardCurrent->trackNo;
               //Get memory real address
               char va[3];   va[2]=NULL;
               va[0]=tempProcess->PCB->ir[2];
               va[1]=tempProcess->PCB->ir[3];
               RAForIO=
                    ADDRESSMAP(tempProcess,atoi(va));
               Task=4;
               StartCH3();
          }

     }
     else     //if PD
     {
          if(  tempProcess->PCB->llc>=
               tempProcess->PCB->tll)
          {
               tempProcess->PCB->ErrorMessage=2;
               ADDpidTo(tempProcess->pid,6);
               DELpidFrom(4);
          }
```

```
else
{
     if(CheckTrack()==1)
     {
          TrackNoForIO=GetTrackFromDrum();
          //update PCB
          if(  tempProcess->PCB->
               OutPutCardHead==NULL &&
               tempProcess->PCB->
               OutPutCardTail==NULL)
          {
               tempProcess->PCB->
               OutPutCardHead=
               (struct CardListNode *)
                    malloc(sizeof(struct
                    CardListNode));
               tempProcess->PCB->
               OutPutCardHead->trackNo=
               TrackNoForIO;
               tempProcess->PCB->
               OutPutCardHead->next=NULL;
               tempProcess->PCB->
               OutPutCardTail=
               tempProcess->PCB->
               OutPutCardHead;
               tempProcess->PCB->
               OutPutCardCurrent=
               tempProcess->PCB->
               OutPutCardHead;
          }
          else
          {
               tempProcess->PCB->
               OutPutCardTail->next=
               (struct CardListNode *)
                    malloc(sizeof(struct
                    CardListNode));
               tempProcess->PCB->
               OutPutCardTail=
               tempProcess->PCB->
               OutPutCardTail->next;
               tempProcess->PCB->
               OutPutCardTail->trackNo=
               TrackNoForIO;
               tempProcess->PCB->
               OutPutCardTail->next=NULL;
          }
```

```
                    //find memory RA
                    char va[3];    va[2]=NULL;
                    va[0]=tempProcess->PCB->ir[2];
                    va[1]=tempProcess->PCB->ir[3];


RAForIO=ADDRESSMAP(tempProcess,atoi(va));

//RAForIO=GetMemoryRAForIO(ptr,va);
                    Task=5;
                    StartCH3();
               }
          }
     }
}
else if(PCBQueueSQHead!=NULL)
{
     tempProcess=NULL;
     tempProcess=
          FindProcessTableNode(PCBQueueSQHead->pid);

     if(CheckFrame()==1)
     {
          //Allocate
          int FrameNo=GetFrameFromMemory();
          //Update PidForFrame[30]
          PidForFrame[FrameNo]=tempProcess->pid;
          //Update page table
          int PageTableFrameNo=
               GetPageTableFrameNo(tempProcess);
          int PageTableLength=
               GetPageTableLength(tempProcess);
          UpdatePageFrameNo(tempProcess,
               tempProcess->PCB->FaultedPage,FrameNo);
          //Adjust TrackForPage[10]
          if(tempProcess->PCB->TrackForPage
               [tempProcess->PCB->FaultedPage]==-1)
          {
               if(CheckTrack()==1)
               {
                    int TrackNo=GetTrackFromDrum();
                    tempProcess->PCB->TrackForPage
                    [tempProcess->PCB->FaultedPage]=
                              TrackNo;
                    UpdatePageTableLength
                    (tempProcess,PageTableLength+1);
               }
          else
          {
               tempProcess->PCB->ErrorMessage=9;
               ADDpidTo(PCBQueueSQHead->pid,6);
               DELpidFrom(5);
               return;
          }
     }
     //Adjust IC,If necessary
     FForSQR=FrameNo;
     Task=7;
     StartCH3();
}
else  //frame not available
{
     //--V-- Victim Process
     //--S-- Swap Queue Head Process
     struct ProcessTableNode *VtempProcess=NULL;
     int VPageTableFrameNo;
     int modified=0;

     int SPageTableFrameNo=
          GetPageTableFrameNo(tempProcess);
     int SPageTableLength=
          GetPageTableLength(tempProcess);

     int VFrameNo=-1;
     //Run page replacement algorithm
     //and Find a victim frame
     while(1)
     {
          VFrameNo=FindVictimFrame();
          VtempProcess=FindProcessTableNode
               (  PidForFrame[VFrameNo]  );
          VPageTableFrameNo=
               GetPageTableFrameNo(VtempProcess);
          if(VFrameNo==VPageTableFrameNo)
          {
               continue;
               VtempProcess=NULL;
          }
          else
               break;
     }
     int VPageFrameNo=-1;
     //Updating Victim Process
     for(int page=0;page<10;page++)
```

```
                {
                if(M[VPageTableFrameNo*BLOCKSIZE+page][0]==
                          '1')
                // if allocated page
                {
                     VPageFrameNo=
                     GetPageFrameNo(VtempProcess,page);
                     if(VPageFrameNo==VFrameNo)
                     {
                          //Update PidForFrame[30]
                          PidForFrame[VFrameNo]=
                          PCBQueueSQHead->pid;
                          //updating victim process
                          M[VPageTableFrameNo*
                               BLOCKSIZE+page][0]='0';
                          if(M[VPageTableFrameNo*BLOCKSIZE
                          +page][1]=='1')
                          {
                               modified=1;
                               M[VPageTableFrameNo*
                               BLOCKSIZE+page][1]='0';
                               }

      M[VPageTableFrameNo*BLOCKSIZE+page][2]=NULL;
      M[VPageTableFrameNo*BLOCKSIZE+page][3]=NULL;
      //updating swap process
      UpdatePageFrameNo
      (tempProcess,tempProcess->PCB->FaultedPage,VFrameNo);
      if(modified==1)
      {
           VFForSQW=VFrameNo;
           TrackNoForSQW=VtempProcess->PCB->
                TrackForPage[page];
      }
      else
           FForSQR=VFrameNo;
           break;
      }//if(atoi(PageFrameNo)==VFrameNo)
 }//if(M[VPageTableFrameNo*BLOCKSIZE+page][0]=='1')
 }
 //Locate Drum Track with faulted page
 if(tempProcess->PCB->TrackForPage
      [tempProcess->PCB->FaultedPage]==-1)
 {
      if(CheckTrack()==1)
      {
           int TrackNo=GetTrackFromDrum();
```

```
          tempProcess->PCB->TrackForPage
          [tempProcess->PCB->FaultedPage]=TrackNo;
          UpdatePageTableLength
               (tempProcess,SPageTableLength+1);
     }
     else
     {
          tempProcess->PCB->ErrorMessage=9;
          ADDpidTo(PCBQueueSQHead->pid,6);
          DELpidFrom(5);
          return;
     }
}
if(modified==1)
     Task=6;
else
     Task=7;
StartCH3();
}  //else frame not available
}//else if(PCBQueueSQHead!=NULL)
}
```

Appendix B

B.1 Data Set for the Workload with Quantum Size=1

PID	ARRIVAL	FINISH	TURNAROUND	WAITING
0	37	78	41	0
1	77	166	89	5
2	128	262	134	19
3	185	330	145	16
5	285	400	115	18
4	241	461	220	41
6	347	534	187	62
7	375	538	163	65
8	414	570	156	49
9	450	658	208	43
13	659	745	86	13
10	502	763	261	48
11	560	830	270	25
12	621	926	305	50
16	848	959	111	29
14	760	1005	245	39
15	812	1035	223	39
17	887	1135	248	42
21	1099	1201	102	17
18	951	1203	252	43
20	1061	1316	255	45
19	998	1334	336	55
23	1264	1395	131	32
22	1224	1452	228	40
24	1338	1481	143	20
25	1378	1559	181	32
27	1479	1650	171	33
26	1426	1667	241	50
29	1582	1670	88	10
28	1538	1739	201	26
32	1774	1865	91	31
31	1728	1867	139	34
30	1688	1868	180	34
33	1812	1994	182	43
34	1861	2134	273	26
37	2017	2162	145	11
35	1914	2164	250	22
39	2173	2294	121	17
36	1972	2333	361	46
40	2207	2346	139	18
38	2065	2450	385	41
41	2247	2494	247	44
45	2517	2579	62	8
43	2367	2582	215	36

42	2316	2619	303	31
44	2425	2683	258	18
47	2642	2788	146	30
48	2678	2810	132	32
46	2592	2840	248	37
49	2730	2943	213	54
53	2933	3028	95	12
51	2835	3046	211	25
50	2780	3064	284	32
56	3127	3212	85	19
55	3081	3240	159	18
54	3041	3286	245	18
52	2893	3335	442	42
57	3164	3480	316	48
58	3237	3520	283	33
61	3390	3563	173	15
59	3283	3608	325	39
63	3565	3716	151	16
60	3353	3730	377	48
64	3597	3793	196	17
62	3517	3819	302	21
65	3645	3886	241	31
67	3787	3999	212	40
69	3920	4043	123	8
70	3940	4127	187	22
66	3741	4141	400	32
71	4040	4143	103	3
68	3850	4211	361	35
72	4072	4258	186	11
73	4156	4397	241	15
75	4256	4451	195	24
77	4354	4489	135	5
74	4206	4495	289	19
76	4313	4570	257	15
79	4528	4687	159	51
80	4564	4709	145	46
78	4480	4771	291	60
81	4602	4795	193	58
82	4652	4905	253	49
85	4844	4906	62	18
83	4709	4966	257	24
84	4806	5025	219	20
87	4983	5131	148	46
88	5019	5153	134	45
86	4943	5202	259	53
89	5057	5275	218	61
90	5106	5360	254	45

91	5158	5387	229	28
93	5307	5405	98	10
92	5220	5457	237	24
94	5400	5547	147	16
95	5450	5605	155	27
97	5548	5659	111	20
96	5510	5700	190	50
98	5598	5734	136	32
99	5639	5736	97	29

B.2 Data Set for the Workload with Quantum Size=2

PID	ARRIVAL	FINISH	TURNAROUND	WAITING
0	37	78	41	0
1	77	173	96	10
2	128	269	141	17
3	184	346	162	18
5	284	382	98	9
4	240	455	215	33
7	374	537	163	54
6	330	561	231	54
8	418	574	156	50
9	472	673	201	48
10	510	754	244	49
11	562	796	234	29
13	663	824	161	13
12	625	882	257	32
15	831	982	151	50
16	875	998	123	43
14	791	1018	227	56
17	916	1114	198	49
21	1124	1257	133	21
19	1021	1275	254	30
20	1079	1327	248	22
18	967	1345	378	43
24	1322	1438	116	12
23	1272	1463	191	23
22	1170	1547	377	33
25	1362	1596	234	33
26	1410	1668	258	36
27	1460	1692	232	21
28	1520	1752	232	17
29	1633	1782	149	6
32	1791	1870	79	25

31	1745	1890	145	25
30	1705	1952	247	42
33	1829	2038	209	52
35	1931	2126	195	35
34	1887	2127	240	45
37	2031	2128	97	8
36	1992	2242	250	23
38	2121	2337	216	35
40	2251	2350	99	31
39	2205	2378	173	37
41	2289	2485	196	54
42	2338	2573	235	30
43	2391	2575	184	26
45	2520	2599	79	9
44	2449	2688	239	29
46	2613	2791	178	36
47	2663	2799	136	42
48	2697	2828	131	36
49	2736	2926	190	42
50	2785	3040	255	28
53	2941	3052	111	13
51	2839	3082	243	25
56	3115	3211	96	15
55	3079	3232	153	18
54	2989	3290	301	42
52	2897	3356	459	47
61	3381	3458	77	7
58	3229	3516	287	9
57	3155	3528	373	37
59	3277	3622	345	14
60	3341	3692	351	20
62	3473	3750	277	32
63	3555	3795	240	25
64	3591	3822	231	33
67	3776	3978	202	10
65	3641	3990	349	28
66	3719	3998	279	16
69	3875	4058	183	3
72	4103	4224	121	40
68	3837	4238	401	45
71	4069	4248	179	36
70	4017	4362	345	52
73	4141	4418	277	54
75	4293	4447	154	13
74	4243	4488	245	13
77	4451	4590	139	22
80	4574	4666	92	39

79	4541	4667	126	44
78	4501	4689	188	52
76	4351	4801	450	66
82	4684	4878	194	21
81	4615	4896	281	44
83	4734	5006	272	40
85	4919	5020	101	10
84	4826	5046	220	18
86	4967	5123	156	14
88	5079	5177	98	14
87	5043	5204	161	25
89	5116	5272	156	27
90	5167	5385	218	35
93	5319	5415	96	25
91	5221	5431	210	35
92	5285	5491	206	29
95	5464	5587	123	15
94	5424	5616	192	22
97	5562	5692	130	29
96	5524	5707	183	33
99	5652	5748	96	30
98	5612	5750	138	22

B.3 Data Set for the Workload with Quantum Size=4

PID	ARRIVAL	FINISH	TURNAROUND	WAITING
0	37	78	41	0
1	77	173	96	9
2	128	262	134	15
3	184	341	157	22
5	285	377	92	10
4	240	421	181	22
8	412	515	103	41
7	380	541	161	49
6	332	601	269	65
9	454	645	191	48
13	668	731	63	5
11	558	822	264	26
10	504	835	331	41
15	758	865	107	15
14	716	900	184	22
12	620	902	282	19
16	909	964	55	0
17	950	1078	128	12
18	999	1165	166	22
19	1049	1254	205	35

21	1149	1266	117	16
20	1112	1316	204	25
22	1203	1337	134	16
23	1293	1429	136	22
24	1329	1454	125	17
25	1368	1523	155	21
26	1416	1612	196	39
27	1471	1653	182	17
29	1573	1681	108	13
32	1732	1817	85	40
28	1537	1818	281	60
30	1626	1828	202	47
31	1688	1833	145	47
33	1772	1944	172	32
34	1886	2037	151	11
37	2034	2127	93	7
35	1936	2139	203	27
36	1994	2222	228	34
39	2142	2264	122	25
40	2176	2301	125	28
38	2088	2302	214	33
41	2237	2437	200	33
43	2338	2480	142	32
42	2287	2526	239	29
45	2472	2616	144	22
47	2561	2679	118	41
48	2595	2701	106	31
46	2521	2713	192	38
44	2397	2804	407	57
49	2636	2948	312	38
50	2701	2963	262	33
51	2750	3019	269	27
53	2850	3093	243	19
52	2814	3152	338	54
56	3082	3183	101	28
55	3032	3220	188	31
54	2982	3285	303	46
58	3175	3422	247	32
57	3120	3424	304	54
61	3325	3517	192	17
59	3237	3546	309	36
60	3285	3612	327	22
64	3539	3657	118	12
62	3463	3728	265	41
63	3507	3758	251	31
65	3583	3848	265	30
66	3635	3948	313	34

69	3793	3992	199	6
71	3989	4102	113	25
68	3755	4104	349	32
67	3684	4105	421	37
72	4023	4119	96	23
70	3883	4131	248	30
73	4126	4316	190	13
74	4177	4392	215	19
77	4331	4455	124	9
75	4227	4472	245	24
80	4469	4573	104	23
79	4429	4576	147	24
76	4285	4650	365	40
78	4377	4696	319	36
82	4594	4845	251	34
85	4742	4899	157	27
83	4640	4955	315	32
81	4511	5022	511	53
84	4706	5058	352	31
88	4992	5078	86	7
87	4926	5172	246	27
86	4888	5223	335	31
89	5054	5257	203	25
90	5103	5343	240	35
93	5254	5377	123	5
91	5151	5519	368	35
95	5384	5536	152	14
92	5214	5539	325	21
94	5304	5621	317	21
97	5558	5670	112	3
99	5650	5746	96	33
98	5608	5747	139	33
96	5442	5762	320	37

B.4 Data Set for the Workload with Quantum Size=5

PID	ARRIVAL	FINISH	TURNAROUND	WAITING
0	37	78	41	0
1	77	173	96	9
2	128	262	134	15
3	184	341	157	22
5	285	377	92	10
4	240	421	181	22
8	412	515	103	41
7	380	541	161	49
6	332	601	269	65

9	454	645	191	48
13	668	731	63	5
11	558	822	264	26
10	504	835	331	41
15	758	865	107	15
14	716	900	184	22
12	620	902	282	19
16	909	964	55	0
17	950	1078	128	12
18	999	1165	166	22
19	1049	1254	205	35
21	1149	1266	117	16
20	1112	1316	204	25
22	1203	1337	134	16
23	1293	1429	136	22
24	1329	1454	125	17
25	1368	1523	155	21
26	1416	1612	196	39
27	1471	1653	182	17
29	1573	1681	108	13
32	1732	1817	85	40
28	1537	1818	281	60
30	1626	1828	202	47
31	1688	1833	145	47
33	1772	1944	172	32
34	1886	2037	151	11
37	2034	2127	93	7
35	1936	2139	203	27
36	1994	2222	228	34
39	2142	2264	122	25
40	2176	2301	125	28
38	2088	2302	214	33
41	2237	2437	200	33
43	2338	2480	142	32
42	2287	2526	239	29
45	2472	2616	144	22
47	2561	2679	118	41
48	2595	2701	106	31
46	2521	2713	192	38
44	2397	2804	407	57
49	2636	2948	312	38
50	2701	2963	262	33
51	2750	3019	269	27
53	2850	3093	243	19
52	2814	3152	338	54
56	3082	3183	101	28
55	3032	3220	188	31

54	2982	3285	303	46
58	3175	3422	247	32
57	3120	3424	304	54
61	3325	3517	192	17
59	3237	3546	309	36
60	3285	3612	327	22
64	3539	3657	118	12
62	3463	3728	265	41
63	3507	3758	251	31
65	3583	3848	265	30
66	3635	3948	313	34
69	3793	3992	199	6
71	3989	4102	113	25
68	3755	4104	349	32
67	3684	4105	421	37
72	4023	4119	96	23
70	3883	4131	248	30
73	4126	4316	190	13
74	4177	4392	215	19
77	4331	4455	124	9
75	4227	4472	245	24
80	4469	4573	104	23
79	4429	4576	147	24
76	4285	4650	365	40
78	4377	4696	319	36
82	4594	4845	251	34
85	4742	4899	157	27
83	4640	4955	315	32
81	4511	5022	511	53
84	4706	5058	352	31
88	4992	5078	86	7
87	4926	5172	246	27
86	4888	5223	335	31
89	5054	5257	203	25
90	5103	5343	240	35
93	5254	5377	123	5
91	5151	5519	368	35
95	5384	5536	152	14
92	5214	5539	325	21
94	5304	5621	317	21
97	5558	5670	112	3
99	5650	5746	96	33
98	5608	5747	139	33
96	5442	5762	320	37

References

[1] Shaw, Allan C., “The logical design of operating systems”, *Prentice Hall*, 1974.

[2] Henry, G. J. “The Fair Share Scheduler”, *Bell System Technical Journal*, October, 1984.

[3] J. Kay and P. Lauder, “A Fair Share Scheduler”, *Communications of ACM*, 31(1), Jan. 1998.

[4] Larmouth, J. “Scheduling for a Share of the Machine”, *Software Practice and Experience*, vol 5, 1975, pp29-49.

[5] Larmouth, J. “Scheduling for Immediate Turnaround”, *Software Practice and Experience*, vol 8, 1978, pp559-578.

[6] Newbury, J.P., ”Immediate Turnaround-An Exclusive Goal”, *Software Practice and Experience*, vol 8, 1982.

[7] Woodside, C.M. “Controllability of Computer Performance Tradeoffs Obtained Using Controlled-Share Queue Schedulers” *IEEE Trans. On Software Engineering*, Vol. SE-12, no.10, October, 1986.

[8] Sharma, Onkar P., "Enhancing Operating System Course Using a Comprehensive Project: Decades Of Experience Outlined," *CCSC: Eastern Conference*, PP.206-213, 2007.

[9] Onkar Sharma et al, "An Operating System Project, its accompanying problems, and their object-oriented design solution", *The Journal of Computing for Small Colleges*, Vol. 8, No. 2, Nov. 1992.

[10] William Stallings, *Operating systems*, Prentice Hall, fourth edition, 2001.

[11] Jason Nieh, Chris Waill, and Hua Zhong, “Virtual-time round robin: An O(1) proportional share scheduler”, In *Proceedings of the 2001 USENIX Annual Technical Conference*, June 2001.

[12] Andrew S. Tanenbaum and Albert S. Woodhull, *Operating systems: Design and implementation*, Prentice Hall, second edition, 1997.

Bibliography

[1] A. Silberschatz, P. Baer Galvin, G. Gagne, Operating System Concepts, *John Wiley & Sons*, 6th edition.

[2] Andrew S. Tanenbaum, Modern Operating Systems, Second Edition, *Prentice Hall of India*.

[3] Zhao, Xiaodong, “Process Scheduling in Linux”, Department of Computer Science, Helsinki University.

[4] Hideaki Takagi, “Exact Analysis of round-robin Scheduling of Services”, *IBM J. Res. Development*. Vol 31, July 1987.

[5] A. Demers, S. Keshav, and S. Shenker, “Analysis and Simulation of a Fair Queueing Algorithm”, *in Proceedings of ACM SIGCOMM ’89*, Austin, TX, Sept. 1989, pp 1-12

[6] M. Jones, D. Rosu, and M. Rosu, “CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities,” *in Proceedings of the 16th Symposium on Operating Systems Principles*, ACM press, New York, Oct. 1997, pp. 198-211

[7] H. Custer, Inside Windows NT, Redmond, WA, USA: Microsoft Press 1993

[8] M. Beck, H. Bohme, M. Dziadzka, and U. Kunitz, Linux Kernel Internals, Reading, MA: Addison-Wesley, 2nd ed., 1998.