Parametric Optimization Of Some Critical Operating System Functions – An Alternative Approach To The Study Of Operating Systems Design

Tarek M. Sobh, Abhilasha Tibrewal

Department of Computer Science and Engineering
University of Bridgeport
Bridgeport, CT 06601, USA

Abstract

Operating systems theory primarily concentrates on the optimal use of computing resources. The study of operating systems design and concepts by way of parametrically optimizing critical operating system functions is the focus of this work. Our work marks a new approach to teaching and studying operating systems design processes. The four specific operating system functions studied are those of CPU scheduling, memory management, deadlock/synchronization primitives and disc scheduling. The aim of the study is to first introduce and discuss the modules in light of previous research, discuss in details the affecting parameters and their interaction and attempt to optimize some of the lesser-established parameter-performance relationships by way of simulations. Results of the simulations of the four functions are then analyzed in light of specific parameters and the effect they have on the overall system performance. System performance is judged by many measures, including: average turn around time, average waiting time, throughput, CPU utilization, fragmentation, response time, and several other module specific performance measures.

Some of the parameters studied in the CPU scheduling module include: the round robin time slot, aging parameters, preemption switches and context switching time. Simulation of multilevel feedback queues is attempted and the performance is judged in terms of the above mentioned performance measures. In the context of memory management, some of the parameters studied include: memory size, RAM and disc access times, compaction thresholds, memory placement algorithm choice, page size and the time quantum value. The attempted simulation uses the continuous memory scheme. In the deadlock/synchronization module, the parameters studied include: the total number of processes, the total number of available resources and the maximum number of resources required by the processes. Four deadlock handling mechanisms are discussed and a deadlock avoidance algorithm is simulated. The denial (rejection) rate of requests for resources quantifies system performance. Within the disc-scheduling module, the parameters studied include: disc configuration/size, disc access time, disc scheduling algorithm choice, disc writing mechanism and all the parameters utilized in the memory management module. Performance is judged in terms of the above mentioned performance parameters and also the percentage seek and latency times. Some of the simulation specific results tend to highlight the role of optimizing the value of the round robin quantum in the modules, the importance of average seek and average latency times versus the system performance and the comparative performance of the various memory placement algorithms and disc scheduling algorithms. Lastly, an attempt to integrate the four specified modules is discussed to attain the final goal of designing an optimal operating system with the right permutation of design parameters to achieve excellent performance measures for various process mixes.

1. Introduction

The intended focus of the proposed research is to study operating systems design and concepts by way of parametrically optimizing critical operating systems functions. CPU scheduling, memory management, deadlock/synchronization primitives and disc scheduling are the four specific functions under scrutiny. The study proposes to introduce all the above and provide an in-depth discussion of the involved parameters. All the concerned

parameters will be elaborated upon, focusing on their effect on system performance as well as interaction with other parameters. The study also evaluates certain parameters of each module whose effect on system performance is not yet well established. Finally, the modules are discussed from an integrated perspective.

2. Background

The operating system is an essential part of any computer system, the operating system being the program that acts as an intermediary between a user of the computer and the computer hardware. The operating system has also been termed as the resource allocator. Much of the operating-system theory concentrates on the optimal use of computing resources. One important goal for an operating system is to make the computer system convenient to use and another goal is to use the computer hardware in an efficient manner [3]. The focus in this work is on the efficiency aspect.

The development of operating system over the past 40 years has evolved from batch systems to time shared operating systems. Spooling and multiprogramming were some important concepts in the development of the latter systems. In these time-sharing operating systems, several jobs must be kept simultaneously in memory, which requires some form of memory management; the requisition of an on-line file-system which resides on a collection of discs necessitates disc management; the need for concurrent execution mechanism requires sophisticated CPU scheduling schemes; and the need to ensure orderly execution demands job synchronization and deadlock handling [3].

2.1. Processes And Process Control Block

At the heart of the operating system is the process mix. A process is a program in execution. As a process executes, it changes state, which is defined by that process's current activity. A process may be in a new, ready, running, waiting or terminated state. Each process is represented in the operating system by its own process control block (PCB) [1]. Figure 1 shows typical process mix and Table 1 illustrates an instance of a process mix.

Figure 1. A Typical PCB

- Process ID (PID)
- Arrival Time
- Execution Time
- Priority
- Size
- Location
- Program Counter Value
- Registers / Threads
- Needed Resources

Table 1. A Sample Process Mix

Process ID	Arrival Time	Priority	Execution Time
1	0	20	10
2	2	10	1
3	4	58	2
4	8	40	4
5	12	30	3

A PCB includes the following fields:

- Process ID (PID): The unique identifier used by other processes for scheduling, communication and any other purpose.
- Arrival Time: The time at which the process enters the process queue for scheduling purposes.
- Estimated Execution Time: Used by scheduling algorithms that order processes by execution time.
- Priority / Process Type: Used by scheduling algorithms that follow priority-based criterion.
- Size: The size of the process in bytes.
- Location: The memory location of a process.
- Program Counter Value: The address of next instruction to be executed.
- Registers / Threads: The state of different registers used by processes
- Needed Resources: Indicates the quantities/types of system resources needed by a process.

In other words, a Process Control Block is a data structure that stores certain information about each process [1].

2.2. Performance Parameters

Quantifying performance is essential to optimization. Following are some of the common parameters used to benchmark performance.

- CPU Utilization: The ratio of time that the CPU is doing actual processing to the total CPU time observed. This is a true measure of performance since it measures the efficiency of the system. An idle CPU has 0% CPU utilization since it offers null performance per unit cost. The higher the CPU utilization, the better the efficiency of the system.
- Turnaround Time: The time between a process's arrival into the system and its completion. Two related parameters that can be studied include the average turnaround time and maximum turnaround time. The turnaround time includes the context switching times and execution times. The turnaround time is inversely related to the system performance, i.e. lower turnaround times imply better system performance.
- Waiting Time: Waiting time is the sum of the periods spent waiting in the ready queue. The CPU scheduling algorithm does not affect the execution time of a process but surely determines the waiting time. Mathematically, it is the difference between the turnaround time and execution time. Like turnaround time, it inversely affects the system performance and has two related forms: average waiting time and maximum waiting time.
- Throughput: The average number of processes completed per unit time. Even though this is a reasonable measure of operating system performance, it should not be the sole performance criterion taken into account. This is so because throughput does not take into account loss of performance caused by starvation. In the case of starvation, the CPU might be churning out completed processes at a very high rate but there might be a process stuck in the scheduler with an infinite wait time. Higher throughput is generally considered as indicative of increased performance.
- Response Time: The time difference between submission of the process and the first I/O operation. It affects performance inversely. However, it is not considered to be a good measure and is rarely used.

2.3. Evaluation Technique

When developing an operating system or the modules thereof, evaluation of its performance is needed before it is installed for real usage. Evaluation provides useful clues to which algorithms would best serve different cases of application [4]. There are several evaluation techniques. Lucas (1971, as cited in [4]) summarized and compared some frequently used techniques, including cycle and times, instruction mixes, kernels, models, benchmarks, synthetic programs, simulation, and monitor. All techniques can be basically classified into three types: the analytic method, implementation in real time systems, and the simulation method.

In the analytic method, a mathematical formula is developed to represent a computing system. This method provides clear and intuitive evaluation of system performance, and is most useful to a specific algorithm. However, it is too simple to examine a complex and real system.

Another technique is to implement an operating system in a real machine. This method produces a complete and accurate evaluation. One of the disadvantages of this technique is the dramatic cost associated with the implementation. In addition, evaluation is dependent on the environment of the machine in which the evaluation is carried out.

Simulation is a method that uses programming technique to develop a model of a real system. Implementation of the model with prescribed jobs shows how the system works. Furthermore, the model contains a number of algorithms, variables, and parameters. By changing these factors in the simulation, one is able to know how the system performance would be affected and, therefore, to predict possible changes in the performance of the real system. This method has a reasonable complexity and cost. It was viewed as the most potentially powerful and flexible of the evaluation techniques (Lucas, 1971 as cited in [4]).

The model for a full simulation of an operating system contains numerous parameters. Identification of the most important parameters in terms of system performance is useful for a complete evaluation and for a fair design of a real system [4].

2.4. Purpose Of The Study

This study proposes to present an alternative approach to the study of operating systems design by way of parametrically optimizing critical operating systems functions. This shall entail detailed discussions of the four tasks of CPU scheduling, synchronization and deadlock handling, memory management and disc scheduling in terms of the involved parameters. In addition, it is also proposed to use the simulation technique to analyze some of the stated parameters in their respective modules:

- CPU scheduling: round robin time quantum, aging parameters, α-values and initial execution time estimates, preemption switches, context switching time.
- Synchronization and Deadlock Handling: total number of processes, total number of available resources, maximum number of resources required by the processes, rejection rate over time.
- Memory Management: memory size, RAM and disc access times, compaction thresholds, memory placement algorithms, page size, page replacement algorithms, time quantum value, fragmentation percentage in time windows over time.
- Disc scheduling: disc configuration/size, disc access time, disc scheduling algorithms, disc writing mechanisms and all the above mentioned memory management parameters.

System performance shall be judged by many measures, including: average turnaround time, average waiting time, throughput, CPU utilization, fragmentation, response time, and several other module specific performance measures. Finally, it is proposed to discuss the integration of the four tasks into an optimal operating systems using the right permutation of design parameters.

3. **Parametric Optimization of Operating Systems Modules**

At the onset, this section presents a general outline of the methodology involved. Module-wise simulations include the description of the specific method of data collection.

Each of the proposed four tasks of the operating system: CPU scheduling, synchronization and deadlock handling, memory management and disc scheduling are described with emphasis on the involved parameters. The parameters are discussed in terms of their interaction with the operating system function under study and their resultant effect on the system performance.

A simulation technique is used to evaluate system performance in all the four modules. It is specifically used to explore the effect of parameters whose relation with system performance is not proportional. Evaluation of system performance against these parameters is conducted by analyzing a number of sample runs of the respective simulated modules.

Every simulated module generates a random process mix. Assuming that there are six parameters in a specific module and each parameter can take ten possible values, the total number of possible permutations becomes one million (10x10x10x10x10x10). Furthermore, these one million permutations are applicable to the particular process mix only. Therefore, each run of a specific simulated module uses the same process mix in our case. This enables the analysis of the studied parameter versus performance measures to have a uniform base for comparisons. An exhaustive study of all possible permutations is beyond the scope of this study. Moreover, the purpose of this study is to provide an alternative approach to studying operating systems design. Hence, we include optimization of some parameters in each module to serve as a model example.

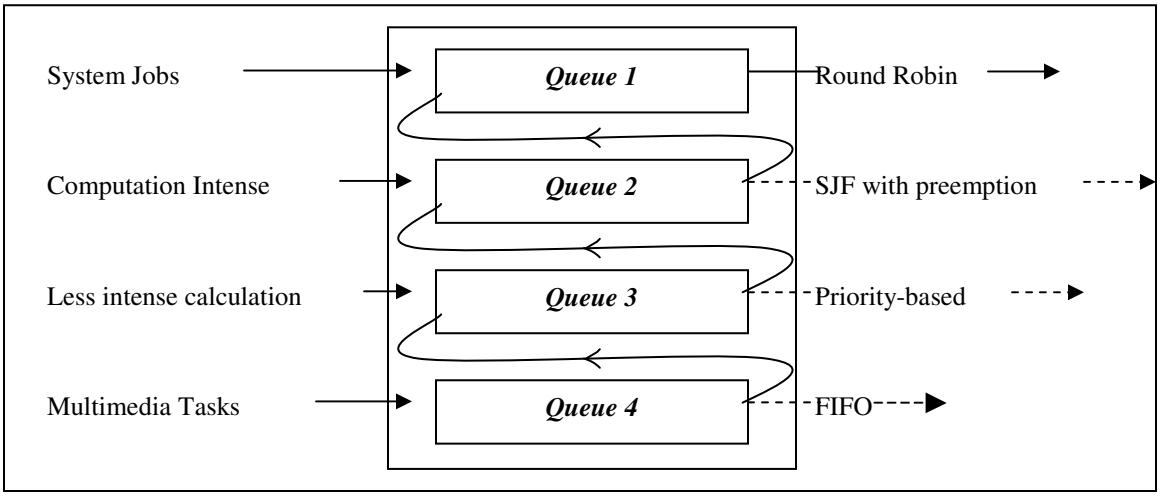
Module specific methodology is included within the respective module and contains detailed information about the independent and dependent variables. The independent variables include the studied parameters in each of the operating system functions while the performance measures like percentage CPU utilization, average turnaround time, average waiting time, throughput, fragmentation percentage, rejection/denial rate, percentage seek time and percentage latency time constitute the dependent variables.

Next, we elaborate on a module wise discussion of the four studied operating system functions, namely: CPU scheduling, synchronization and deadlock handling, memory management and disc scheduling. At the end of this section, the integration of the four modules into an optimal operating system is explained.

3.1. CPU Scheduling

An operating system must select processes (programs in execution) for execution in some order. The selection process is carried out by an appropriate scheduling algorithm. CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms, for example, first come first served, shortest job first, priority, round-robin schemes.

Figure 2. A Multi-Level Feedback Queue



Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups/types. A multilevel queue-scheduling algorithm (see Figure 2) partitions the ready queue into several separate queues. The processes are assigned to a queue, generally based on some property of the process. Each queue has its own scheduling algorithm.

Processes are assigned to a queue depending on their type, characteristics and priority. Queue 1 gets processes with maximum priority such as system tasks and Queue 4 gets processes with the lowest priority such as non-critical audio/visual tasks. The idea is to separate processes with different CPU-burst characteristics.

Each queue has a different scheduling algorithm that schedules processes for the queue. Processes in Queue 2 get CPU time only if Queue 1 is empty. Similarly, processes in Queue 3 receive CPU attention only if Queue 1 and Queue 2 are empty and so forth.

However, if the above-described method is implemented as is, processes in queues 2, 3 and 4 have a potential of starvation in case Queue 1 receives processes constantly. To avoid this problem, aging parameters are taken into account. Aging means that processes are upgraded to the next queue after they spend a pre-determined amount of time in their original queue. For example, a process spends a pre-determined amount of time unattended in Queue 4 will be moved to Queue 3. Processes keep moving upwards until they reach Queue 1 where they are guaranteed to receive CPU time (or execute in other queues before reaching Queue 1).

In general, a multilevel feedback queue scheduler is defined by the number of queues, the scheduling algorithm for each queue, the method used to assign the entering processes to the queues and the aging parameters.

Although a multilevel feedback queue is the most general scheme, it is also the most complex and has the potential disadvantage of high context switching time.

Many of the scheduling algorithms use execution time of a process to determine what job is processed next. Since it is impossible to know the execution time of a process before it begins execution, this value has to be estimated. α, a first degree filter, is used to estimate the execution time of a process as follows:

$z_n = \alpha z_{n-1} + (1 - \alpha)\, t_{n-1}$

where, z is estimated execution time
t is the actual time
α is the first degree filter and $0 \leq \alpha \leq 1$
The following example provides a deeper understanding of the issue at hand.

Table 2. Calculating Execution Time Estimates

Processes	z_n	t_n
P_0	10	6
P_1	8	4
P_2	6	6
P_3	6	4
P_4	5	17
P_5	11	13
P_6	12	….

Here,
$\alpha = 0.5$
$z_0 = 10$

Then by formula,
$z_1 = \alpha z_0 + (1-\alpha) t_0$
$= (0.5) (10) + (1-0.5) (6)$
$= 8$
and similarly z_2, z_3….z_6 are calculated.

Thus, an estimated execution time for the first process is assumed and then the filter is used to make further estimations (see Table 2). However, the choice of the value of α affects the estimation process. Following is the scenario when α takes the extreme values:

- $\alpha = 0$ means that z_n does not depend on z_{n-1} and is equal to t_{n-1}
- $\alpha = 1$ means that z_n does not depend on t_{n-1} and is equal to z_{n-1}

Table 3. α-updating scheme

z_n	t_n	Square Difference
10	6	
$(\alpha)\ 10 + (1-\alpha)\ 6 = 6 + 4\alpha$	4	$[(6+4\alpha) - 4]^2 = (2+4\alpha)^2$
$(6+4\alpha)\alpha + (1-\alpha)\ 4 = 4\alpha^2+2\alpha+4$	6	$[(4\alpha^2+2\alpha+4) - 6]^2 = (4\alpha^2+2\alpha-2)^2$

Consequently, we start with a symbolic value of α and obtain f (α) i.e. the sum of square difference (see Table 3). Further, differentiation of this and equating it to zero gives the value of α for which the difference between the actual time and estimated time is minimum. The following exemplifies α-update in the above example.
In the above example, at the time of estimating execution time of P3, we update α as follows.

The sum of square differences is given by,
$SSD = (2+4\alpha)^2 + (4\alpha^2+2\alpha-2)^2 = 16\alpha^4 + 16\alpha^3 + 4\alpha^2 + 8\alpha + 8$
And, d/dx [SSD] = 0 gives us,
$8\alpha 3 + 6\alpha 2 + \alpha + 1 = 0$ (Equation 1)
Solving Equation 1, we get $\alpha = 0.7916$.
Now,
$z_3 = \alpha z_2 + (1-\alpha)\ t_2$
Substituting values, we have
$z_3 = (0.7916)\ 6 + (1-0.7916)\ 6$
$= 6$
We shall now discuss the parameters involved in a CPU scheduler using the multilevel feedback queue algorithm.

3.1.1. Parameters Involved

Parameters that influence the system performance are hereby enumerated:

- Time slot for the round robin queue (Queue 1)

- Aging time for transitions from Queue 4 to Queue 3, Queue 3 to Queue 2 and Queue 2 to Queue 1, i.e. the aging thresholds for FIFO, priority-based and SJF queues
- α-values and initial execution time estimates for the FIFO, SJF and priority-based queues.
- Choice of preemption for the SJF and Priority based queues.
- Context switching time

Effect of Round Robin Time Slot: The choice of the round robin queue can make the performance vary widely. For example, a small time quantum results in higher context switching time, which in turn translates to low system performance in form of low CPU utilization, high turnaround times and high waiting times. On the other hand, a big time quantum results in FIFO behavior with effective CPU utilization, lower turnaround and waiting times but with the potential of starvation. Thus, finding an optimal time slot value becomes imperative for maximum CPU utilization with lowered starvation problem.

Effect of Aging Thresholds: A very large value for the aging thresholds makes the waiting and turnaround times unacceptable. These are signs of processes nearing starvation. On the other hand, a very small value makes it equivalent to one round robin queue. Zhao [6] enumerates the aging parameters of 5, 10 and 25 for the SJF, Priority-based and FIFO queues respectively as the optimal aging thresholds for the specified process mix. Some of the process mix specifications being: process size vary from 100KB to 3MB; estimated execution time range from 5 to 35ms; priority values vary from 1 to 4; memory size is 16MB; disc drive configuration is 8 surfaces, 64 sectors and 1000 tracks.

Effect of α-values and initial execution time estimates: Su [4] has studied the effect of prediction of burst time on system performance of a simulated operating system as part of her study. She has used an α update scheme as was previously discussed. For her specified process mix, she reports that the turnaround time obtained from predicted burst time is significantly lower than the one obtained by randomly generated burst time estimates. The α value is recomputed/updated after a fixed number of iterations.

Effect of choice of preemption: Preemption undoubtedly increases the number of context switches, and increased number of context switches inversely affects the efficiency of the system. However, preemptive scheduling has been shown to decrease waiting and turnaround time measures in certain instances [3]. There are two preemption switches involved in this module, one for the SJF queue (Queue 2) and the other for the priority-base queue (Queue 3). In SJF scheduling, the advantage of choosing preemption over non-preemption is largely dependent on the CPU burst time predictions, but that is a difficult proposition in itself.

Effect of Context Switching Time: An increasing value of context switching time inversely affects the system performance in an almost linear fashion. The context switching time tends to affect system performance inversely. As the context switching time increases, so does the average turnaround and average waiting time. The increase of the context switching time pulls down the CPU utilization.

In keeping with the above discussion, the simulation of the above module and the analysis of the collected data focus on the optimal round robin time quantum and effect of the α updating scheme.

3.1.2. Simulation Specifications and Method of Data Collection

The implemented multi-level feedback queue scheduler consists of four linear queues, the first is FIFO, the second queue is priority-based, the third one is SJF and the fourth (highest priority) is round robin. Feedback occurs through aging; aging parameters differ, i.e., each queue has a different aging threshold before a process can migrate to a higher priority queue. Processes are assigned to one of the queues upon entry. A process can migrate between the various scheduling queues based on the aging parameter of the queue it was initially assigned.

Round robin time quantum, the preemptive switches for the SJF and priority-based queues, aging parameters for the SJF, priority-based and FIFO queues, context switching time, initial execution time estimates and α values for the FIFO, SJF and priority queues are some of the independent variables in this module. To optimize any one of them, we need to keep every other parameter fixed and vary the studied parameter. We have attempted to optimize the round robin time and the effect of the α update scheme to serve as a model. Thus, the round robin time was the

variable parameter in our case and all other parameters were fixed parameters. The dependent variables of the module are the performance measures: average turnaround time, average waiting time, CPU utilization and throughput.

(a) Snapshot at time step 1
SCHEDULING STARTED
Scheduling Started

Total Process Created = 5

Process Number = 1
Queue Number = 2
Execution Time = 1
Priority = 3
Arrival Time = 1

(b) Snapshot at time step 1

FifoQ Content : 5
PriorityQ Content : 1 4
SJFQ Content : 3
RRQ Content : 2
DoneQ Content :

Process# 2 is executing.

(c) Snapshot at time step 2
Total Process Created = 3

FifoQ Content : 5 7
PriorityQ Content : 1 4
SJFQ Content : 3
RRQ Content : 2 6 8
DoneQ Content :

Process# 2 is executing.

(d) Snapshot at time step 3

Total Process Created = 2

FifoQ Content : 5 7 9
PriorityQ Content : 1 4
SJFQ Content :
RRQ Content : 6 8 2 3 10
DoneQ Content :

Process# 6 is executing.

(e) Snapshot at time step 4
Total Process Created = 4

FifoQ Content : 7 9 14
PriorityQ Content : 5 11 13
SJFQ Content : 1 4
RRQ Content : 8 2 3 10 6 12
DoneQ Content :

Process# 8 is executing.

(f) Snapshot at time step 5
Total Process Created = 1

FifoQ Content : 9 14
PriorityQ Content : 5 11 13 7 15
SJFQ Content : 1 4
RRQ Content : 2 3 10 6 12
DoneQ Content : 8

Process# 2 is executing.

Figure 3. Snapshot of process mix at time steps 1-5

Data was collected by means of multiple sample runs. The output from the sample run indicates a timeline, i.e. at every time step, it indicates which processes are created (if any), which ones are completed (if any), processes which aged in different queues. The following excerpts from an output file (see Figure 3) illustrate the aging of process 1 from the priority based queue to the SJF queue (the aging parameter for Queue 3 was set to be 3 in this run). Figure 3, part (a) shows process mix snapshot at time step 1. Five processes are created at this instance and the PCB parameters for process number 1 are displayed. Part (b) illustrates the contents of the queue at this time step. Process 1 is assigned to the priority queue. Given an aging parameter of 3 for the priority queue, process 1 should migrate to the SJF queue at time step 4 unless it finishes execution before that. Snapshots at time step 2 (part (c)) and time step 3 (part (d)) show that process 2 and process 6 get CPU attention since they are in the round robin queue (queue with highest priority). Therefore, process 1 does not get the opportunity to execute and migrates to the SJF queue at time step 4 (part (e)). Part (f) illustrates the completion of process 8 and inclusion of the same in the done queue. A complete walkthrough of this sample run for the CPU scheduling module is available at www.bridgeport.edu/~sobh/SampleRun1.doc

3.1.3. Simulation Results and Discussion

Table 4 and the corresponding charts (Figure 4 (a) – (d)) illustrate the effect of varying the round robin quantum time over the various performance parameters. This parameter plays a critical role as, whenever present, it is the processes in this queue that are being scheduled for execution.

It can be clearly seen from the Table 4 how the time slot of the round robin queue affects the various performance parameters. While the throughput is observed to be inversely proportional, the other three performance measures seem to be directly proportional. In other words, with increasing the time slot the round robin queue moves towards the behavior of a FIFO queue with high average turnaround times and average waiting times. The throughput decreases but the percentage CPU utilization improves at a steady rate.

Table 4. Effect of Round Robin Time Slot on the Performance Parameters

RRTimeSlot	Av.Turnaround Time	Av. Waiting Time	CPU Utilization	Throughput
2	19.75	17	66.67 %	0.026
3	22.67	20	75.19 %	0.023
4	43.67	41	80.00 %	0.024
5	26.5	25	83.33 %	0.017
6	38.5	37	86.21 %	0.017

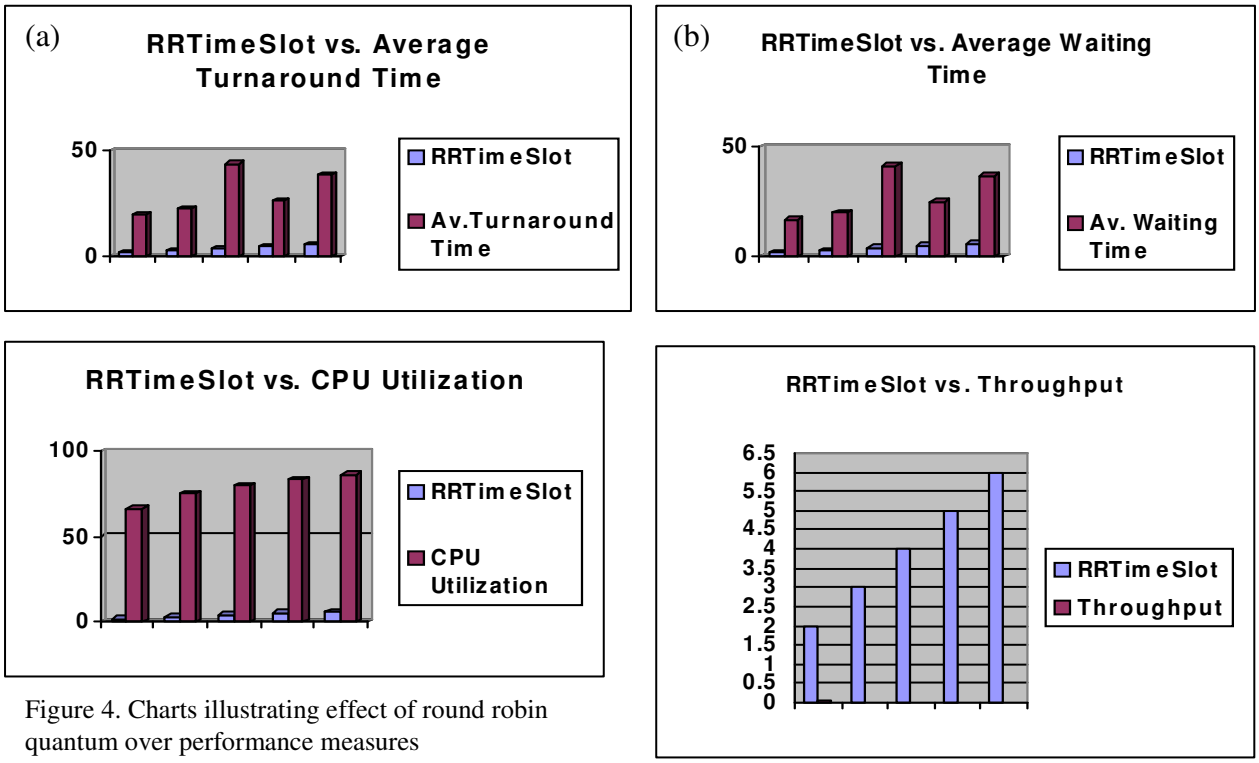


Figure 4. Charts illustrating effect of round robin quantum over performance measures

Since the round robin is the highest priority queue in the multilevel feedback queue scheduler, it has the greatest influence over the scheduler performance. With CPU Utilization of 80% and throughput of 0.024, time slot value of 4 time units comes out to be the optimal value in this simulation for the specific process mix.

Next we illustrate the effect of α updating on the system performance. Table 5 compares the performance measures as the value of round robin time slot is varied with α updated at regular intervals. The performance measure values in the bracket are the corresponding values when the α updating scheme was not implemented.

Table 5. Comparing performance measures of a CPU scheduler with α-update and one with no α-update (the values for the scheduler with no α-update is in brackets)

RRTimeSlot	Av.Turnaround Time	Av. Waiting Time	CPU Utilization	Throughput
2	19.75 (19.75)	17 (17)	66.67 (66.67) %	0.026 (0.026)
3	22.67 (22.67)	20 (20)	75.19 (75.19)%	0.023 (0.023)
4	43.67 (43.67)	41 (41)	80.00 (80.00)%	0.024 (0.024)
5	26.5 (26.5)	25 (25)	83.33 (83.33)%	0.017 (0.017)
6	38.5 (38.5)	37 (37)	86.21 (86.21)%	0.017 (0.017)

As is evident from Table 5, α updating did not affect system performance in our case. Again, the result is specific for our particular process mix.

To summarize, it is the optimal value of the round robin quantum along with smallest possible context switching time that tends to maximize performance in context of CPU scheduling in our simulation. α-updating did not tend to affect performance.

3.2. Synchronization And Deadlock Handling

To ensure the orderly execution of processes, the operating system provides mechanisms for job synchronization and communication, and ensures that jobs do not get stuck in a deadlock, forever waiting for each other.

Synchronization: Synchronization problems arise because sections of code that constitute the critical sections overlap and do not run atomically. A critical section is a part of a process that accesses shared resources. Two processes should not enter their critical sections at the same time, thus preventing the problem. To run a critical section atomically means that the section is executed either as a whole or not at all [5]. Once the critical section of a process begins, it must be completed or rolled back. Thus, important points to note about critical sections are:

- Critical sections of code must be run atomically
- Mutual exclusion must be ensured of more than one critical section

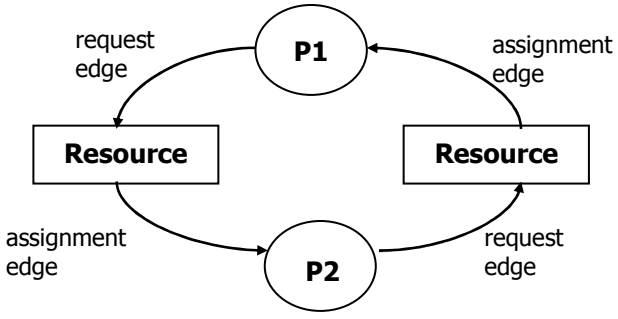
The above is ensured by using pieces of code that block a process that attempts to run its critical section while the critical section of another process is being executed. The process is then unblocked when the other process's critical section completes execution. This is known as Sleep and Wakeup. The above can be implemented by using semaphores, monitors and message passing. Table 6 shows a comparison between the three synchronization methods.

Table 6. A Comparison of three Synchronization Methods

Implementation	Synchronization	Mutual Exclusion	Advantages	Disadvantages
Semaphores	√	√	•	• Low-level implementation • Can cause deadlock
Monitors	√	√	• High level implementation	•
Message Passing	√	√	•	•

Deadlock Handling: A deadlock state is a state of indefinite wait by one or more processes for an event that can be triggered only by one of the waiting processes. Such an event is commonly termed as a resource. The resource could be tangible such as an I/O resource or intangible e.g. shared data [5]. Figure 5 shows a visual representation of a deadlock where process 1 is holding resource 2 and requesting resource 1 and process 2 is holding resource 1 and requesting resource 2.

Figure 5. Visual representation of a deadlock



Four necessary conditions for occurrence of deadlock are:

- Mutual exclusion of resources: Inability of a resource to be used by more than one process
- Hold and wait: A process holds a resource while waiting for another one
- No preemption: The system is incapable of grabbing a resource from a process
- Circular wait.

Algorithms to tackle deadlock conditions are based on the idea of prevention, detection and recovery,

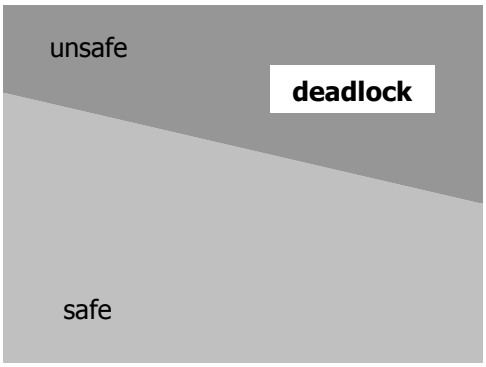
avoidance and ignoring the condition. Ignoring deadlocks is by far the most widely used method.
Deadlock prevention is a set of methods for ensuring that at least one of the necessary conditions cannot hold. Of the four necessary conditions, mutual exclusion cannot be stopped since it is the nature of the process. Hold and wait can be stopped by never assigning a resource to a process unless all the other needed resources are available. However, it suffers from possible starvation of processes. Moreover, resource utilization may be low, since many of the resources may be allocated but unused for a long period. The necessary condition of no preemption can be stopped by using the following rule. If a process that is holding some resource requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted to stop this condition. As evident, it also suffers from potential starvation. One way to ensure that the circular-wait condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration. But the stated method turns out to be expensive in terms of time complexity.

In the environment of handling deadlocks using detection and recovery, the system must provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

A deadlock detection and recovery algorithm involves the following steps: reducing a process-resource graph to a wait-for graph; finding cycles in the graph; determining non-redundant cycles; and finally determining the minimum number of preemptions required. However, it is important to realize that detection and recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm, but also the potential losses inherent in recovering from deadlock.

Figure 6. Safe and Unsafe State



A deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that there can never be a circular-wait condition. The resource allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes. A state is safe if the system can allocate resources to each process in some order and still avoid deadlock. More formally, a system is in a safe state only if there exists a safe sequence. If no safe sequence exists, then the state is said to be unsafe and has the potential of a deadlock situation (see Figure 6). The following example illustrates this deadlock handling method.

Consider a system with five processes P_0 through P_4 and a resource with 10 instances. Table 7 shows the snapshot of the system at time T_0.

Table 7. Snapshot of system at time T_0

Process #	Current Allocation	Maximum Needs
P_0	3	7
P_1	1	3
P_2	3	9
P_3	1	2
P_4	0	4

Now, we need to decide if it is a safe state. At this time, it is a safe state since the safe sequence <P_1, P_3, P_4, P_0, P_2> exists. Suppose now that P_4 requests an instance of the resource. We pretend that this request has been fulfilled and arrive at the new state shown in Table 8.

Table 8. Snapshot of new state pretending request of P_4 is granted

Process #	Current Allocation	Maximum Needs
P_0	3	7
P_1	1	3
P_2	3	9
P_3	1	2
P_4	1	4

Now, we must determine if this new system state is safe. Since the safe sequence <P_3, P_1, P_4, P_0, P_2> exists, thus, the state is safe and the request of P_4 can be granted. However, if P_2 requests an instances of the resource and we pretend the request has been granted, then the new state configuration will be as shown in Table 9.

Table 9. Snapshot of new state pretending request of P_2 is granted

Process #	Current Allocation	Maximum Needs
P_0	3	7
P_1	2	3
P_2	3	9
P_3	1	2
P_4	0	4

This state, however, is an unsafe state since there is no sequence in which the instances of the resource may be granted such that the needs of all the processes are met. Hence, the request of P_2 is not granted in the first place.

In a real system, there are multiple resources, each with one or more instances. Thus, implementing this algorithm in a real system would entail maintaining the table of resources for each resource. Likewise, there could be multiple requests at a given time instance for the resources present, and a deadlock avoidance algorithm would have to determine whether each of the requests would put the state in a safe or unsafe condition. It is evident that this algorithm proves to be an expensive one.

In a subsequent subsection, our simulation emulates this deadlock avoidance algorithm using one resource.

3.2.1. Parameters Involved

The parameters worthy of study in this module are:

- Total number of processes
- Total number of available resources
- Maximum number of resources required by the processes

Effect of number of processes: An increase in the number of processes will increase the number of requests for resources.

Effect of number of available resources: The more the number of available resources, the more requests can be granted, since for the same requirement there are more resources available.

Effect of maximum number of resources required by the processes: This parameter is directly linked to the number of requests and thus, determines the demand for the resources required. The more the resources required, the more the probability of declined requests.

3.2.2. Simulation Specifications and Method of Data Collection

The deadlock avoidance module uses one resource type. The input is a table of resources that specifies the current usage of the resource and the maximum usage of the resource process-wise and a resource request by one of the processes. The output is either a granted request or a rejected request depending on whether the same generates a safe or unsafe state in terms of deadlock.

Performance in this module is quantified by means of rejection rate or denial rate. It is the percentage of rejected or denied requests over the total number of requests. A higher rejection rate by all means is a sign of poor system performance.

The total number of processes, the total number of resources and the maximum resource requisitions of the process are the independent variables here. The effect of each one is studied on the dependent variable of rejection/denial rate separately while keeping the other ones fixed. Rejection/denial rate over time is also studied. Sample runs of the module with brief parameter-wise analysis are available at www.bridgeport.edu/~sobh/SampleRun2.doc

3.2.3. Simulation Results and Discussion

Rejection Rate versus Total Number of Processes
Fixed Parameters for this part:
Total Number of Resources: 6
Maximum Resources per process:

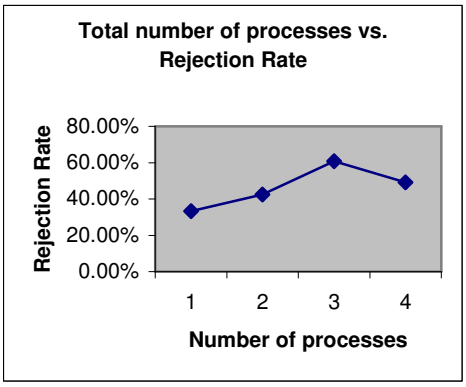
Process	1	2	3	4	5	6	7
Resources	5	4	5	4	5	4	5

Table 10. Total number of processes vs. Rejection Rate

Total Number of Processes	4	5	6	7
Rejection Rate	33.33%	42.5%	60.87%	49.2%
Total Number of Requests	27	40	69	63

Table 10 shows the collected data and Figure 7 shows the corresponding graph. The statistics show a trend of increasing rejection rate as the number of processes increases with everything else kept constant. The number of requests made also increases and influences the same too.

Figure 7. Total number of processes vs. rejection rate



Rejection Rate versus Total Number of Resources
Fixed Parameters for this part:
Total Number of Processes: 4
Maximum Resources per process:

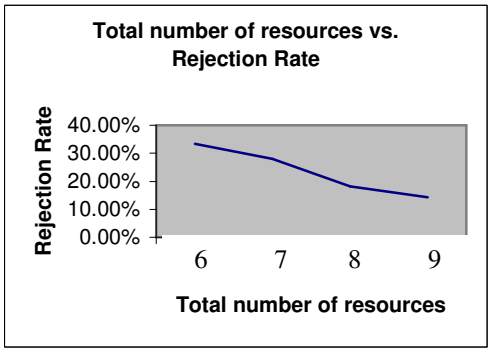
Process	1	2	3	4
Resources	5	4	5	4

Table 11. Total number of resources vs. Rejection Rate

Total Number of Resources	6	7	8	9
Rejection Rate	33.33%	28%	18.18%	14.29%
Total Number of Requests	27	25	22	21

Table 11 shows the collected data and Figure 8 shows the corresponding graph. The statistics clearly indicate that as the number of available resources increases, more requests are successfully granted. The increase in the number of granted requests make the rejection rate move down steadily. In other words, an increase in the total number of resources tends to affect the rejection rate in a linear fashion.

Figure 8. Total number of resources vs. rejection rate



Rejection Rate versus Maximum Number of Resources Per Process
Fixed Parameters for this part:
Total Number of Processes: 4
Total Number of Resources: 6

Table 12. Maximum Resources needed per process vs. Rejection Rate

Distribution of Maximum Number of Resources (MaxR) per Process (P)	Rejection Rate	Total Number of Requests
P 1 2 3 4 MaxR 5 4 5 4	33.3%	27
P 1 2 3 4 MaxR 3 4 5 4	20%	20
P 1 2 3 4 MaxR 2 4 5 4	16.67%	18
P 1 2 3 4 MaxR 2 3 5 4	12.5%	16

Figure 9. Maximum resources needed per process vs. rejection rate

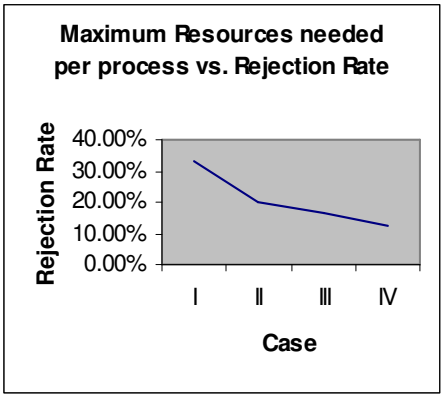


Table 12 shows the collected data and Figure 9 shows the corresponding graph. The statistics indicate that as the number of maximum resources needed for a process decreases so does the rejection rate. The total number of requests made is directly proportional to the maximum resources needed for a process. A decrease in the latter decreases the former and consequently decreases the rejection rate in a linear manner.

Rejection Rate over Time

Fixed Parameters for this part:
Total Number of Processes: 5
Total Number of Resources: 10
Maximum Resources per process:

Process	1	2	3	4	5
Resources	7	5	9	2	4

Table 13. Rejection rate over time

Time Window (5 time units)	Rejection Rate (%)
0 to 5	0
5 to 10	0
10 to 15	20
15 to 20	60
20 to 25	60
25 to 30	80
30 to 35	60
35 to 40	0

Figure 10. Rejection rate over time

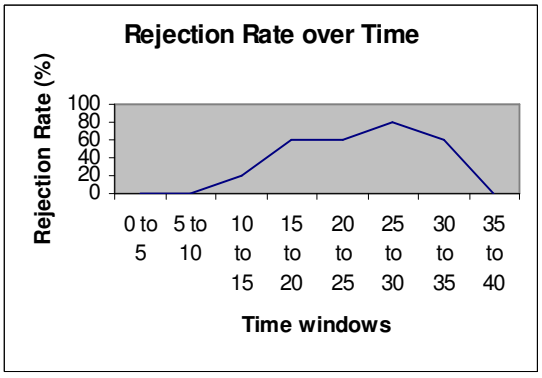


Table 13 shows the collected data and Figure 10 shows the corresponding graph. As the simulation starts, initially all requests are granted, as the resources are available and the rejection rate is null at this point. As more resources are allocated, the available resource number lowers and thus process requests are rejected if they lead to an unsafe state. Rejection increases as more and more resources are allocated. Then comes the phase when enough resources have been granted to meet the maximum need of some processes. At this time, these processes whose maximum resource needs have been met start releasing the resources. Thus, the available resources start increasing and consequently the denial rate decreases finally coming back to the state of null rejection. As is evident from Figure10, the shape of the rejection rate over time graph closely resembles a normal curve.

To summarize, the rejection rate is controlled by the dynamic mix of the number of processes, the number of available requests as well as the maximum resource requirement per process. In addition, another crucial determining factor is the order in which the requests are made. More available resources and fewer resource requests improve performance in a linear manner. However, if the number of maximum resource requirement per process exceeds the number of available resources, deadlock is inevitable. On the other hand, if the number of resources available is at least equal to the sum of the maximum resource requirement per process, the system can boast of a null rejection rate. The rejection rate over time curve closely resembles a normal curve.

3.3. Memory Management

Memory is an important resource that must be carefully managed. The part of the operating system that manages memory is called the memory manager. Memory management primarily deals with space multiplexing. All the processes need to be scheduled in such a way that all the users get the illusion that their processes reside on the RAM. Spooling enables the transfer of a process while another process is in execution. The job of the memory manager is to keep track of which parts of memory are in use and which parts are not in use, to allocate memory to processes when they need it and deallocate it when they are done, and to manage swapping between main memory and disc when main memory is not big enough to hold all the processes.

Three disadvantages related to memory management are:

- the synchronization problem
- the redundancy problem
- the fragmentation problem

The first two are discussed below and the fragmentation problem is elaborated upon a little later.

Spooling, as stated above, enables the transfer of one or more processes while another process is in execution. It aims at preventing the CPU from being idle, thus, managing CPU utilization more efficiently. The processes that are being transferred to the main memory can be of different sizes. When trying to transfer a very big process, it is possible that the transfer time exceeds the combined execution time of the processes in the RAM. This results in the CPU being idle which was the problem for which spooling was invented. This problem is termed as the **synchronization problem**. The reason behind it is that the variance in process size does not guarantee synchronization.

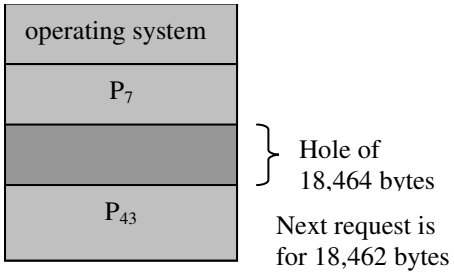
The combined size of all processes is usually much bigger than the RAM size and for this very reason processes are swapped in and out continuously. The issue regarding this is the transfer of the entire process when only part of the code is executed in a given time slot. This problem is termed as the **redundancy problem**.

There are many different memory management schemes. Memory management algorithms for operating systems range from the single user approach to paged segmentation. Some important considerations that should be used in comparing different memory management strategies include hardware support, performance, fragmentation, relocation, swapping, sharing and protection. The greatest determinant of any method in a particular system is the hardware provided.

Fragmentation, Compaction and Paging: **Fragmentation** is encountered when the free memory space is broken into little pieces as processes are loaded and removed from memory. Fragmentation can be internal or external.

Consider a hole of 18,464 bytes as shown in Figure 11. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach is to allocate very small holes as part of the larger request. Thus, the allocated memory may be slightly larger then the requested memory. The difference between these two numbers is **internal fragmentation** – memory that is internal to a partition, but is not being used [3]. In other words, unused memory within allocated memory is called internal fragmentation [2].

Figure 11. Internal fragmentation



External fragmentation exists when enough total memory space exists to satisfy a request, but it is not contiguous; storage is fragmented into a large number of small holes. In Figure 12 two such cases can be observed. In part (a), there is a total external fragmentation of 260K, a space that is too small to satisfy the requests of either of the two remaining processes, P4 and P5. In part (c), however, there is a total external fragmentation of 560K. This space would be large enough to run process P5, except that this free memory is not contiguous. It is fragmented into two pieces, neither one of which is large enough, by itself, to satisfy the memory request of process P5. This

Figure 12. External Fragmentation

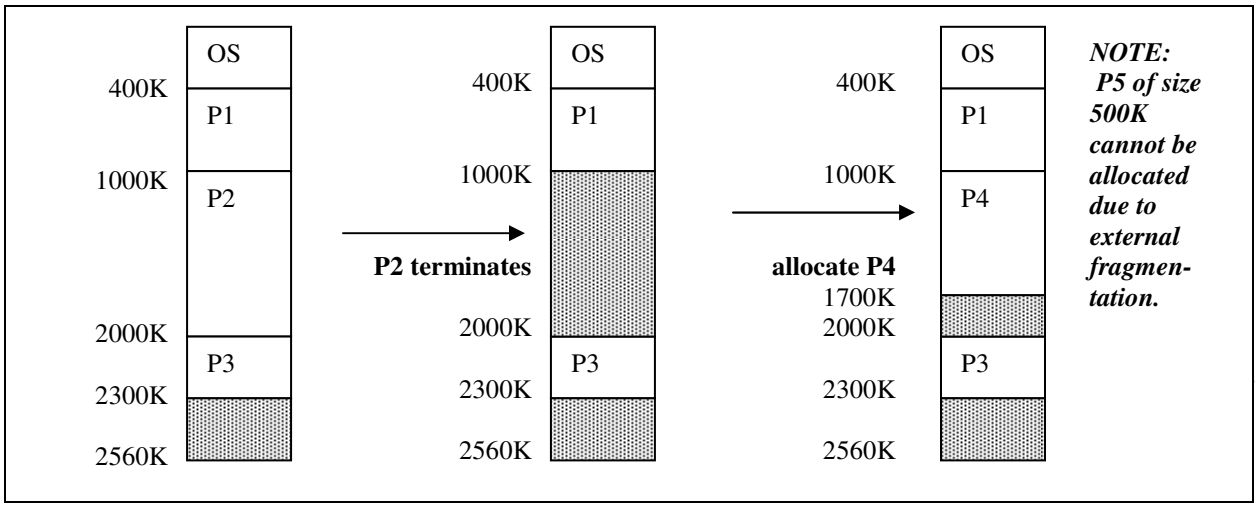
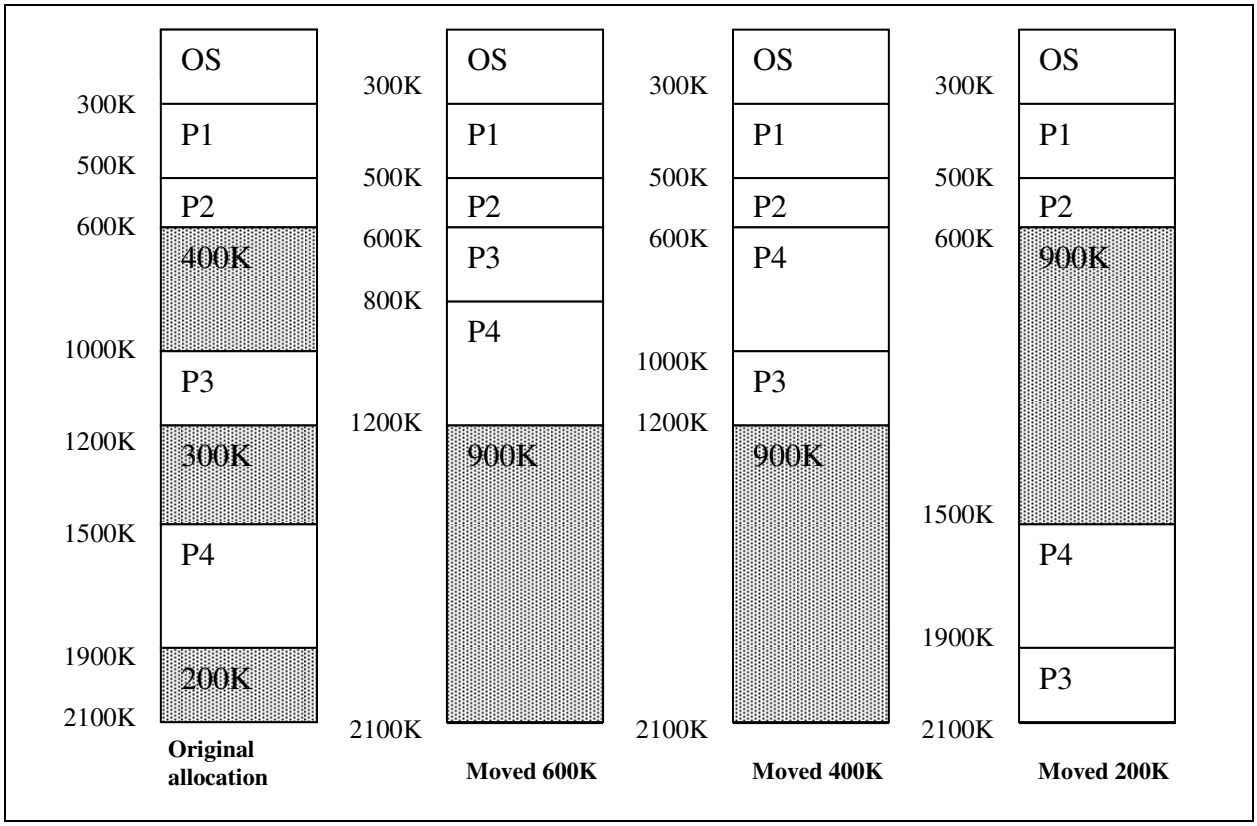


Figure 13. Different ways to compact memory



Compaction is usually defined by the following two thresholds:

- Memory hole size threshold: If the sizes of all the holes are at most a predefined hole size, then the main memory undergoes compaction. This predefined hole size is termed as the hole size threshold. For example, if we have two holes of size 'x' and size 'y' respectively and the hole threshold is 4KB, then compaction is done provided x <= 4KB and y<=4KB.
- Total hole percentage: The total hole percentage refers to the percentage of total hole size over memory size. Only if it exceeds the designated threshold, compaction is undertaken. Taking the two holes with size 'x' and size 'y' respectively, total hole percentage threshold equal to 6%, then for a RAM size of 32MB, compaction is done only if (x + y)> = 6% of 32MB.

Another possible solution to the external fragmentation problem is to permit the physical address space of a process to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available. One way of implementing this solution is through the use of a **paging** scheme. We discuss paging in greater details a little later in this section.

Memory Placement Algorithms: A fitting algorithm determines the selection of a free hole from the set of available holes. First-fit, best-fit, and worst-fit are the most common strategies used to select a free hole.

- First-fit: Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search is ended. Searching stops as soon as a large enough free hole is found.
- Best-fit: Allocate the smallest hole that is big enough. The entire list needs to be searched, unless the list is kept ordered by size. This strategy produces the smallest leftover hole.
- Worst-fit: Allocate the largest hole. Again, the entire list has to be searched, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

If memory is lost due to internal fragmentation, the choice is between first fit and best fit. A worst fit strategy truly makes internal fragmentation worse. If memory is lost due to external fragmentation, careful consideration should be given to a worst-fit strategy [2].

3.3.1. Continuous Memory Allocation Scheme

The continuous memory allocation scheme entails loading of processes into memory in a sequential order. When a process is removed from main memory, new processes are loaded if there is a hole big enough to hold it. This algorithm is easy to implement, however, it suffers from the drawback of external fragmentation. Compaction, consequently, becomes an inevitable part of the scheme.

3.3.1.1. Parameters Involved

Some of the parameters that influence the system performance in terms of memory management are hereby enumerated:

- Memory size
- RAM access time
- Disc access time
- Compaction algorithms
- Compaction thresholds – Memory hole-size threshold and total hole percentage
- Memory placement algorithms
- Round robin time slot (in case of a pure round robin scheduling algorithm)

Effect of Memory Size: As anticipated, the greater the amount of memory available, the higher would be the system performance.

Effect of RAM and Disc access time: The higher the values of the access times, the lower the time it would take to move processes from main memory to secondary memory and vice-versa thus increasing the efficiency of the operating system. Disc access time is composed of three parts seek time, latency time and transfer rate. The RAM

access time plays a crucial role in the cost of compaction. Compaction entails accessing each byte of the memory twice, thus, the faster the RAM access, the lower would be the compaction times.

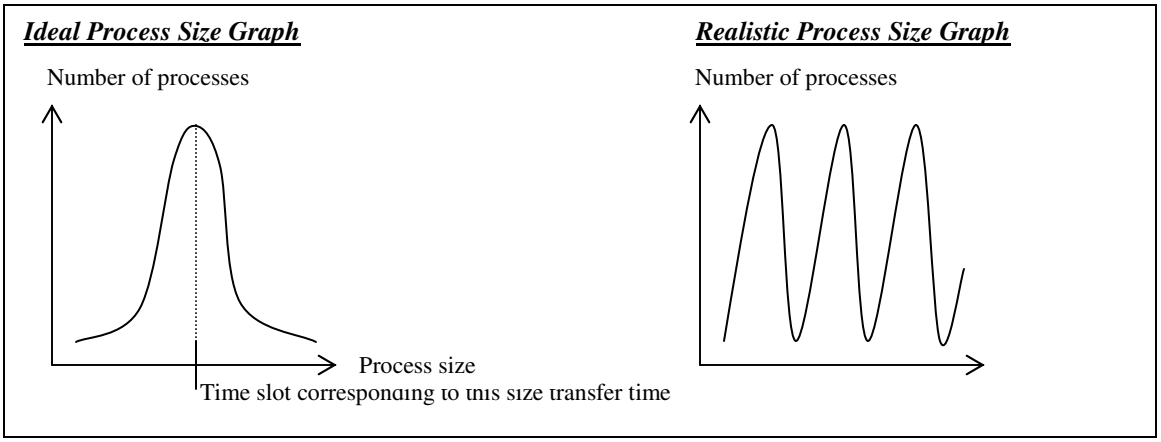
Effect of Compaction Algorithms: Choosing an optimal compaction algorithm is critical in minimizing compaction cost. However, selecting an optimal compaction strategy is quite difficult.

Effect of the Compaction Thresholds: The effect of compaction thresholds on system performance is not as straightforward and has seldom been the focus of studies in this field. Optimal values of hole size threshold largely depend on the size of the processes since it is these processes that have to be fit in the holes. Thresholds that lead to frequent compaction can bring down performance at an accelerating rate since compaction is quite expensive in terms of time.

Effect of Memory Placement Algorithms Silberschatz and Galvin in [3] state that simulations have shown that both first-fit and best-fit are better than worst-fit in terms of decreasing both time and storage utilization. Neither first-fit nor best fit is clearly best in terms of storage utilization, but first-fit is generally faster.

Effect of Round Robin Time Slot: Best choice for the value of time slot would be corresponding to transfer time for a single process (see Figure 14). For example, if most of the processes required 2ms to be transferred, then a time slot of 2ms would be ideal. Hence, while one process completes execution, another has been transferred. However, the transfer times for the processes in consideration are seldom a normal or uniform distribution. The reason for the non-uniform distribution is that there are many different types of processes in a system. The variance as depicted in Figure 14 is too much in a real system and makes the choice of time slot a difficult proposition to decide upon.

Figure 14. Ideal Process Size Graph and Realistic Process Size Graph



In keeping with the above discussion, the simulation of the above module and the analysis of the collected data focus on the optimal round robin time quantum, the memory placement algorithms and fragmentation percentage as a function of time.

3.3.1.2. Simulation Specifications and Method of Data Collection

The attempted simulation implements a memory manager system. The implemented system uses a continuous memory allocation scheme. This simulation uses no concept of paging whatsoever. Round robin mechanism is the scheme for process scheduling.

Following are the details of the involved independent variables:
Fixed parameters:

- Memory Size (32 MB)
- Disc access time (1ms (estimate for latency and seek times) + (job size (in bytes)/500000) ms)

- Compaction threshold (6% and hole size = 50KB)
- RAM Access Time (14ns)

Variable parameters:

- Fitting algorithm (a variable parameter – First Fit, Best Fit, Worst Fit)
- Round Robin Time Slot (a variable parameter, multiple of 1ms)

In addition to the above enumerated parameters, the process sizes range from 20KB to 2MB (multiple of 10KB) and the process execution times vary from between 2 ms to 10 ms (multiple of 1ms). The disc size is taken as 500MB and is half filled with jobs at the beginning of the simulation.

In context of memory management, compaction is the solution for fragmentation. However, compaction comes at its own cost. Moving all holes to one end is an expensive operation. To quantify this parameter, percentage of compaction time against total time is a performance measure that has been added in this module. This measure along with all the other performance measures constitutes the dependent variables in this module.

Data was collected by means of multiple sample runs. A walkthrough of a sample run for this module is available at www.bridgeport.edu/~sobh/SampleRun3.doc

3.3.1.3. Simulation Results and Discussion

The round robin time quantum is one of the two variable parameters studied in this simulation. Table 14 and Figure 15 illustrate the effect of varying the round robin quantum time over the various performance parameters in context of the first fit algorithm.

Table 14. Round Robin Time Quantum vs. Performance Measures

Time Slot	Average Waiting Time	Average Turnaround Time	CPU Utilization	Throughput Measure	Memory fragmentation percentage
2	3	4	5%	5	29%
3	4	4	2%	8	74%
4	5	6	3%	12	74%
5	12	12	1%	17	90%

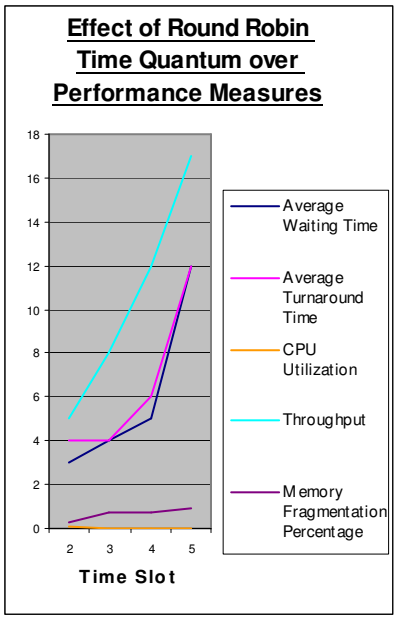


Figure 15. Effect of Round Robin Time Quantum over Performance Measures

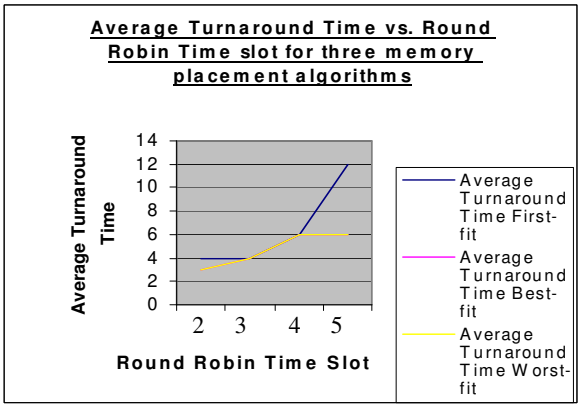
The trends of increasing throughput and increasing turnaround and waiting times are in keeping with round robin scheduling moving towards FIFO behavior with increased time quantum. However, we observe that the CPU utilization is declining with increase in time slot values. This can be attributed to the expense of compaction. Analyzing the fragmentation percentage, it looks like a time slot value of 2 time units is particularly favorable to the same.

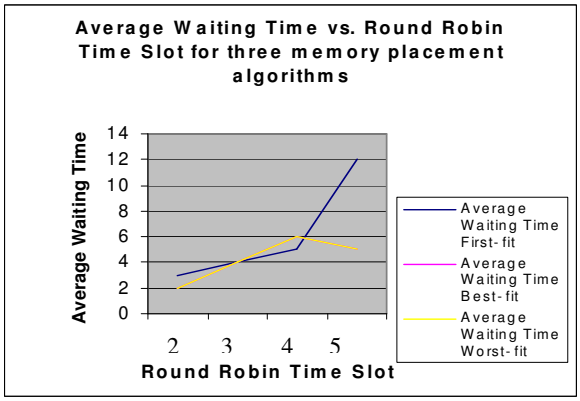
The simulation data collected to compare the three memory placement algorithms by studying the effect of varying round robin time slot over the performance measures for each of the algorithms is given in Table 15 and Figure 16((a) to (e)).

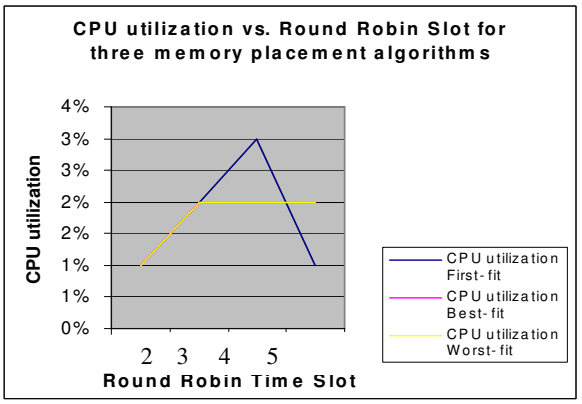
For our particular process mix, best-fit and worst-fit memory placement algorithms gave identical results. None of the memory placement algorithms emerged as a clear winner. However, best-fit and worst-fit algorithms seemed to give more stable fragmentation percentage in the simulations. The aspect of first-fit being faster did not surface in

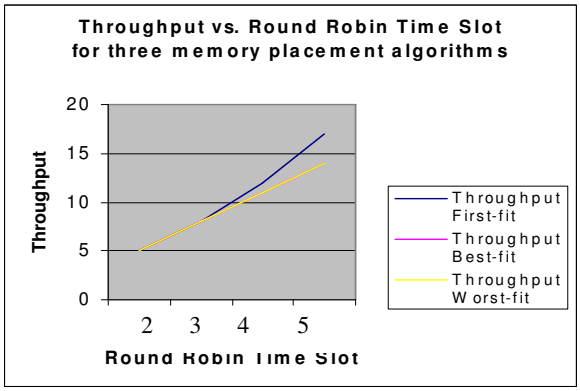
Table 15. Comparing Memory Placement Algorithms

RR Time Slot	Average Turnaround Time			Average Waiting Time			CPU Utilization			Throughput			Fragmentation%		
	First fit	Best fit	Worst fit	First fit	Best fit	Worst fit	First fit	Best fit	Worst fit	First fit	Best fit	Worst fit	First fit	Best fit	Worst fit
2	4	3	3	3	2	2	1%	1%	1%	5	5	5	82	74	74
3	4	4	4	4	4	4	2%	2%	2%	8	8	8	74	74	74
4	6	6	6	5	6	6	3%	2%	2%	12	11	11	74	74	74
5	12	6	6	12	5	5	1%	2%	2%	17	14	14	90	79	79









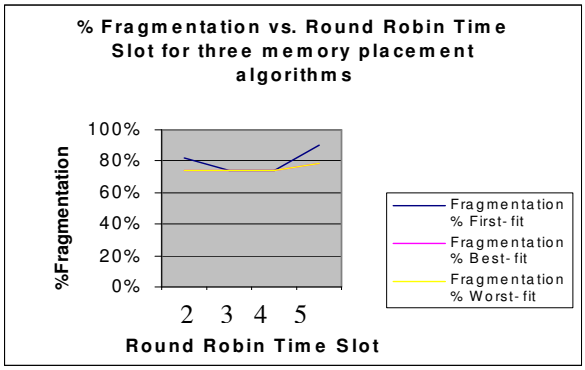


Figure 16. Comparing Memory Placement Algorithms
(a) Average Turnaround time
(b) Average Waiting Time
(c) CPU utilization
(d) Throughput
(e) % Fragmentation

the results due to the nature of the implementation. In the implementation, the worst-fit and best-fit algorithms scan the hole list in one simulated time unit itself. In reality, however, scanning entire hole list by best-fit and worst-fit would make them slower that first-fit, which needs to scan the hole list only as far as it takes to find the first hole that is large enough.

Fragmentation percentage in a given time window over the entire length of the simulation was also studied. The entire simulation was divided into twenty equal time windows and the fragmentation percentage computed for each of the time windows. The trend was studied for four different values of round robin time slot. Since our total hole size percentage threshold was specified as 6%, time frames with fragmentation percentage values higher than that were candidates for compaction [see Table 16 and Figure 17].

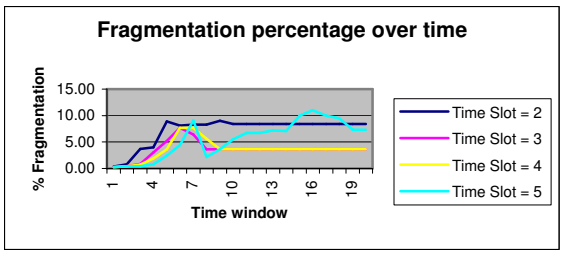
Table 16. Fragmentation percentage over time

Time Window	%Fragmentation			
	Time Slot = 2	Time Slot = 3	Time Slot = 4	Time Slot = 5
1	0.34	0.30	0.27	0.27
2	0.79	0.45	0.45	0.41
3	3.70	0.85	0.73	0.45
4	4.00	3.00	1.90	0.79
5	8.90	5.20	3.60	2.40
6	8.10	7.70	7.70	4.40
7	8.30	6.40	7.70	9.10
8	8.30	3.60	5.60	2.20
9	9.00	3.60	3.60	3.60
10	8.40	3.60	3.60	5.50
11	8.40	3.60	3.60	6.70
12	8.40	3.60	3.60	6.70
13	8.40	3.60	3.60	7.20
14	8.40	3.60	3.60	7.10
15	8.40	3.60	3.60	10.00
16	8.40	3.60	3.60	11.00
17	8.40	3.60	3.60	10.00
18	8.40	3.60	3.60	9.50
19	8.40	3.60	3.60	7.30
20	8.40	3.60	3.60	7.30

However, compaction was undertaken in any of the above candidate frames only if the hole size threshold specification was also met. Looking at Figure 16, we can say that while compaction (if done) for time slot values of 3 and 4 was done in time frames 6 and 7, that for time slot value of 5 was undertaken in the latter half of the simulation.

To summarize, two time units emerged as the optimal time quantum value but none of the memory placement algorithms could be termed as optimal. Studying the fragmentation percentage over time gave us the probable time windows where compaction was undertaken.

Figure 17. Fragmentation percentage over time



3.3.2. Paging Scheme

Paging entails division of physical memory into many small equal-sized frames. Logical memory is also broken into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames. In a paging scheme, external fragmentation can be totally eliminated. However, as is illustrated later, paging requires more than one memory access to get to the data. Also, there is the overhead of storing and updating page tables.

In paging, every address generated by the CPU is divided into two parts: a page number and a page offset. The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address. Two of the more significant parameters in a paging scheme are: page size and page replacement algorithms.

We, hereby, discuss a paging example with a 64MB RAM and 2KB page size. 64MB (2^{26}) memory size can be represented by 26 bits. Likewise, a 2KB page can be represented by 11 bits. Thus, for the page table [see Figure 18], 15 bits are needed for the page number and 11 bits for the page offset. Since there are 2^{15} pages, there shall be 2^{15} entries in the page table. Therefore,

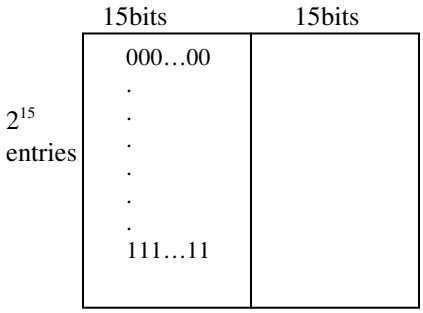
Size of page table = 2^{15} x 30 bits ≈ 123KB

In the above example, if the page size were 1KB, then a 16 bit page number and 10 bit offset would be needed to address the 64MB RAM. In this case,

Size of page table = 2^{16} x 32 bits = 256KB

Consequently, it can be said that a smaller page size results in larger sized page tables and the page table size becomes an overhead itself.

Figure 18. A Page Table



Fragmentation, synchronization and redundancy as discussed in the previous section are three problems that need to be addressed in a memory management setting. In a paging scheme, there is no external fragmentation. However, internal fragmentation exists. Supposing the page size is 2KB and there exists a process with size 72,700 bytes. Then, the process needs 35 pages and 1020 bytes. It is allocated 36 pages with an internal fragmentation of 1028 bytes (2048 – 1020). If the page size were 1KB, the same process would need 70 pages and 1020 bytes. In this case, the process is allocated 71 pages with an internal fragmentation of 4 bytes (1024 – 1020). Thus, a smaller page size is more favorable for reduced internal fragmentation.

In the worst case scenario, a process needs 'n' pages and 1 byte, which results in an internal fragmentation of almost an entire frame. If process size is independent of page size, then

Average internal fragmentation = ½ x page size x number of processes

Hence, it can be observed that a large page size causes a lot of internal fragmentation. On the other hand, a small page size requires a large amount of memory space to be allocated for page tables. One simple solution to the problem of large size page tables is to divide the page table into smaller pieces. One way is to use a two-level paging scheme, in which the page table itself is also paged. However, multilevel paging comes with its own cost – an added memory access for each added level of paging.

Anticipation and page replacement deals with algorithms to determine the logic behind replacing pages in main memory. A good page replacement algorithm has a low page-fault rate. Some common page replacement algorithms are as follows.

Time Stamp Algorithms

- FIFO: A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest is chosen.
- LRU: Least Recently Used (LRU) algorithm associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest time.

Count based Algorithms

- LFU: The least frequently used (LFU) algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.

- MFU: The most frequently used (MFU) algorithm requires that the page with the largest count be replaced. The reason for this selection is that the page with the smallest count was probably just brought in and has yet to be used.

Continuous Memory Allocation versus Paging Allocation
Table 17 gives a comparison between the two studied memory management schemes.

Table 17. Comparing continuous memory allocation scheme with paged allocation scheme

Continuous Memory Allocation Scheme	Paged Allocation Scheme
Advantages: • An easy algorithm for implementation purposes.	Advantages: • No external fragmentation, therefore, no compaction scheme is required.
Disadvantages: • Fragmentation problem makes compaction an inevitable part. Compaction in itself is an expensive proposition in terms of time.	Disadvantages: • Storage for page tables. • Addressing a memory location in paging scheme needs more than one access depending on the levels of paging.

3.3.2.1. Parameters Involved
The new parameters involved in this memory management scheme are:
- Page Size
- Page Replacement Algorithms

Effect of Page Size: A large page size causes a lot of internal fragmentation. This means that, with a large page size, the paging scheme tends to degenerate to a continuous memory allocation scheme. On the other hand, a small page size requires large amounts of memory space to be allocated for page tables. Finding an optimal page size for a system is not easy as it is very subjective dependent on the process mix and the pattern of access.

Effect of Page Replacement Algorithms: Least-recently used, first-in-first-out, least-frequently used and random replacement are four of the more common schemes in use. The LRU is often used as a page-replacement algorithm and is considered to be quite good. However, an LRU page-replacement algorithm may require substantial hardware assistance.
To study the effects of the above parameters on system performance, a new performance measure, namely replacement ratio percentage, is added to the usual list of performance measures. The replacement ratio percentage quantifies page replacements. It is the ratio of the number of page replacements to the total number of page accesses.

3.3.2.2. Implementation Specifics

Though paging was not attempted as part of this study, the implementation specifics of Zhao's study [6] are included here to illustrate one sample implementation.

Zhao, in his study, simulated an operating system with a multilevel feedback queue scheduler, demand paging scheme for memory management and a disc scheduler. A set of generic processes was created by a random generator. Ranges were set for various PCB parameters as follows:
- Process size: 100KB to 3MB
- Estimated execution time: 5 to 35ms
- Priority: 1 to 4

A single level paging scheme was implemented. A memory size of 16MB was chosen and the disc driver configuration: 8 surfaces, 64 sectors and 1000 tracks was used.

Four page replacement algorithms: LRU, LFU, FIFO, random replacement and page size were chosen as the independent variables in context to paging. The dependent variables for the study were average turnaround time and replacement percentage.

3.3.2.3. Implementation Results

The data in Table 18 (taken from Zhao's study [6]) show the effect of replacement algorithms on the replacement ratio.

Table 18. Page Replacement Scheme vs. Replacement Ratio percentage

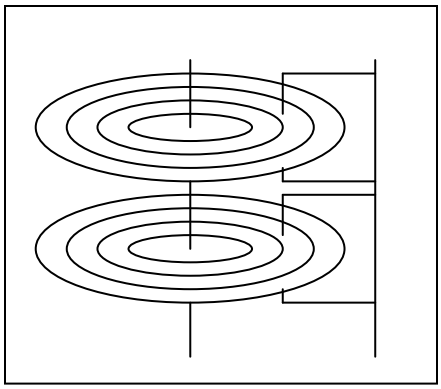
Scheme	**FIFO**	**LRU**	**LFU**	**Random**
Replacement Ratio %	31	30	37	31

After having found the optimal values of all studied parameters except page size in his work, Zhao used those optimal values for 1000 simulations each for a page size of 4KB and 8KB. The latter emerged as a better choice.
In his work, Zhao concludes that 8KB page size and the LRU replacement algorithms constitute the parametric optimization in context to paging parameters for the specified process mix.

3.4. Disc Scheduling Integrated With Ram Manager Module

Disc systems are the major secondary-storage I/O device on most computers. Requests for disc I/O are generated both by the file systems and by virtual-memory systems. Each request specifies the address on the disc to be referenced, which is in the form of a block number. The lower levels of the file-system manager convert this address into the hardware-level partition, cylinder, surface, and sector number.

Figure 19. Hard Disc Structure



One of the functions of the memory manager is to manage swapping between main memory and disc when main memory is not big enough to hold all the processes. The disc, i.e. the secondary storage device, at the same time needs effective management in terms of disc structure [see Figure 19] and capacity, the disc writing mechanism and the scheduling algorithm choice. Since most jobs depend heavily on the disc for program loading and input and output files, it is important that disc service be as fast as possible. The operating system can improve on the average disc service time by scheduling the requests for disc access.

The disc movement is composed of three parts. The three distinct physical operations, each with its own cost, are: seek time, rotational delay/latency time and transfer time.

To access a block on the disc, the system must first move the head to the appropriate track or cylinder. This head movement is called a seek, and the time to complete it is **seek time** [3]. In other words, seek time is the time required to move the access arm to the correct cylinder. The amount of time spent seeking during a disc access depends on how far the arm has to move. If we are accessing a file sequentially and the file is packed into several consecutive cylinders, seeking needs to be done only after all the tracks on a cylinder have been processed, and then the read/write head needs to move the width of only one track. At the other extreme, if we are alternately accessing sectors from two files that are stored at opposite extremes on a disc, seeking is very expensive. Seeking is likely to be more costly in a multiuser environment, where several processes are contending for use of the disc at one time, than in a single-user environment, where disc usage is dedicated to one process. Since it is usually impossible to know exactly how many tracks will be traversed in a seek, we usually try to determine the average seek time required for a particular operation. If the starting and ending positions for each access is random, it turns out that the average seek traverses one-third of the total number of cylinders that the read/write head ranges over [2].

Once the head is at the right track, it must wait until the desired block rotates under the read-write head. This delay is the **latency time** [3]. Hard discs usually rotate at about 5000 rpm, which is one revolution per 12ms. On average, the rotational delay is half a revolution, or about 6ms. As in the case of seeking, these averages apply only when the read/write head moves from some random place on the disc surface to the target track. In many circumstances, rotational delay can be much less than the average [2].

Finally, the actual transfer of data between the disc and main memory can take place. This last part is **transfer time** [3]. The total time to service a disc request is the sum of the seek time, latency time, and transfer time. For most discs the seek time dominates, so reducing the mean seek time can improve the system performance substantially. Thus, the primary concern of disc scheduling algorithms is to minimize seek and latency times. Some of the common disc scheduling algorithms include: FCFS, SSTF, SCAN, C-SCAN, LOOK and C-LOOK algorithms.

3.4.1. Parameters Involved

Parameters that influence the system performance in terms of memory management including disc scheduling are hereby enumerated:

- Disc access time (seek time, latency time and transfer time)
- Disc configuration
- Disc scheduling algorithm
- Disc writing mechanism (where to rewrite processes after processing them in RAM)
- Memory Size
- RAM Access Time
- Compaction thresholds – Memory hole-size threshold and total hole percentage
- Memory placement algorithms
- Round robin time slot

Effect of Disc Access Time: The lower the value of this parameter, the better the system performance. As discussed earlier, seek time, latency time and transfer time together give the disc access time. Since seek is the most expensive of the three operations, lowering seek time is crucial to system efficiency.

Effect of Disc Configuration: Disc configuration relates to the structural organization of the disc into tracks and sectors. Disc surfaces and tracks are determined by hardware specifications. However, some operating systems allow the user to choose the sector size that influences the number of sectors per track. It is an entire sector that is read or written when transfer occurs. This property determines efficiency of many computing algorithms and determines inter-record and intra-record fragmentation in terms of database operations. The number of tracks equals the number of cylinders. Reading and writing on one cylinder reduces the seek time considerably. Thus, even though this parameter does not necessarily affect operating system performance directly, it is nevertheless important to understand its implications.

Effect of Disc Scheduling Algorithm: It is this parameter that primarily determines the possible minimization of seek and latency times. While FCFS algorithm is easy to program and is intrinsically fair, it may not provide the best service. SSTF scheduling substantially improves the average service time but suffers from the inherent problem of starvation of certain processes in case of continuing flow of requests. The SCAN, C-SCAN, LOOK and C-LOOK belong to the more efficient genre of disc scheduling algorithms. These are however, complicated in their respective implementations and are more appropriate for systems that place a heavy load on the disc. With any scheduling algorithm, however, performance depends heavily on the number and types of requests. In particular, if the queue seldom has more than one outstanding request, then all scheduling algorithms are effectively equivalent. In this case, FCFS scheduling is also a reasonable algorithm.

Disc Writing Mechanism: In terms of disc writing mechanism, there is a choice between writing back to where the process was initially read from and writing back to the cylinder closest to the disc head. While the former is straightforward to implement, in no way does it attempt on optimization of seek time. The latter choice, however, results in increased overhead in terms of updating the location of the process every time it is written back to the disc.

Effects of memory size, RAM access times, compaction thresholds, memory placement algorithms and round robin time slot have been discussed in section 3.3.1.1.

In keeping with the above discussion, the simulation of the above module and the analysis of the collected data focus on the optimal round robin time quantum, average seek and latency times. Combination of disc scheduling and the memory placement algorithms also form the subject of the analysis, as does the sector size.

3.4.2. Simulation Specifications and Method of Data Collection

The attempted simulation implements a memory manager system that includes disc scheduling. The implemented system uses a continuous memory allocation scheme. Round robin mechanism is the scheme for process scheduling.

Following are the details of the involved independent parameters:

- Disc access time (seek + latency + (job size (in bytes)/500000) ms where, seek time and latency time are variable parameters)
- Disc configuration (8 surfaces and 300 tracks/surface) – fixed parameter
- Disc scheduling algorithm (a variable parameter – FIFO, SSTF)
- Disc writing mechanism (processes are written at the same place they are read from) - fixed parameter
- Memory Size (32 MB) – fixed parameter
- Compaction threshold (6% and hole size = 50KB) – fixed parameters
- RAM Access Time (14) – fixed parameter
- Fitting algorithm (a variable parameter – First Fit, Best Fit)
- Round Robin Time Slot (a variable parameter, multiple of 1ms)

In addition to the above enumerated parameters, the process sizes range from 20KB to 2MB (multiple of 10KB) and the process execution times vary from 2 ms to 10 ms (multiple of 1ms). The disc size is taken as 600MB and is half filled with jobs at the beginning of the simulation. The jobs are written cylinder-wise on the disc.

In context of disc scheduling, seek times and latency times play a very crucial role in determining the system performance. Hence, two additional parameters, namely, percentage seek time and percentage latency time are added to the array of performance parameters. Both of these parameters are calculated as a percentage of total time. More about seek time and latency time is included along with simulation results. These two, along with the other performance measures, constitute the dependent variables of this simulation.

Data was collected by means of multiple sample runs. A walkthrough of a sample run for this module is available at www.bridgeport.edu/~sobh/SampleRun4.doc. The data was then systematically organized as described.

Initially, the whole data set was divided into four broad categories. The four categories are the four possible combinations of the fitting algorithms (First Fit and Best Fit) and the implemented disc scheduling algorithms (FIFO and SSTF). These categories (FIFO-First Fit, FIFO-Best Fit, SSTF-First Fit, SSTF-Best Fit) were further branched into three groups based on the variable parameter which was chosen amongst average seek time, latency time and round robin time slot. At this level, the performance measures used were: Throughput, %CPU utilization, average turnaround time, average waiting time, % seek time and % latency time. The performance measures were studied as a function of the variable parameter. Performance measures were also studied in terms of sector size for the SSTF-First Fit combination.

3.4.3. Simulation Results and Discussion

At the onset, there is an algorithm combination versus performance analysis that is succeeded by other parameters. These parameters include average seek time, average latency time, round robin time slot and sector size against the performance measures of throughput, %CPU utilization time, average turnaround time, average waiting time, % seek time and % latency time.

Algorithm Combinations vs. Performance Measures
Fixed Parameters:
RR Time Slot: 2ms Average Seek Time: 8ms
Sector Size: 1KB Average Latency Time: 4ms
Simulation Time: 3000ms

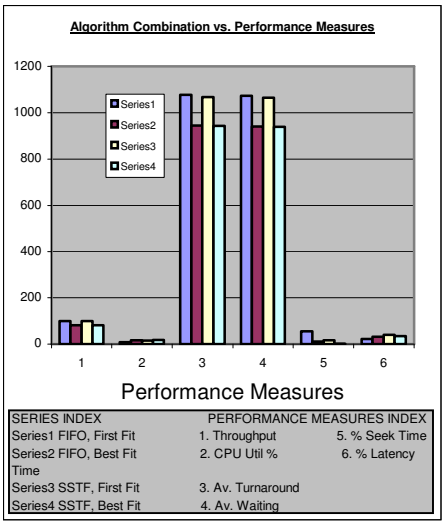
SSTF surely outperforms FIFO in all respects. However, what is interesting to note is that while First Fit algorithm seems to be markedly more efficient in terms of throughput, average turnaround and average waiting time; the best-

fit algorithm tends to outshine percentage CPU utilization and percentage seek time [see Table 19 and Figure 20]. The higher CPU utilization percentage can be explained as a function of lowered seek time.

Table 19. Algorithm Combination vs. Performance Measures

Algorithm Combination	Throughput	%CPU Utilization	Average Turnaround Time	Average Waiting Time	%Seek time	%Latency Time
FIFO, First Fit	99	7.94	1077	1073	55.08	21.48
FIFO, Best Fit	82	16.78	944	940	11.07	31.21
SSTF, First Fit	99	14.85	1068	1064	15.99	40.17
SSTF, Best Fit	82	18.62	943	939	1.32	34.64

Figure 20. Algorithm Combination vs. Performance Measures



From this point on discussion is done focusing on the SSTF-First Fit and FIFO-First Fit algorithm combinations.

Average Seek Time vs. Performance Measures

Measures of seek time include track-to-track, full stroke and average seek time. It is the last that has been used. It is the time (in ms) that the head takes to cover half of the total tracks. It tends to vary between 8 to 10ms. State of the art machines boast of a seek time of 4ms [7]. This explains the average seek time range under scrutiny.

Fixed Parameters:
RR Time Slot: 2ms Simulation Time: 3000ms
Sector Size: 1KB Average Latency Time: 4ms

As evident from the above Tables 20 and 21 and their corresponding graphs [see Figures 21 and 22], average seek time tends to affect CPU utilization. Higher average seek times relate to higher seek time % which pulls down the % CPU utilization. As a special note, the total latency time is unchanged (e.g. in all cases it was 2748ms in case of SSTF-First Fit). It is the increasing seek time that pulls down the % latency time. It should not be misunderstood that higher average seek times improve latency times.

The variable parameter tends to show little if any effect on the other performance measures.

Table 20. Average Seek Time vs. Performance Measures (FIFO–First Fit)

Avg Seek Time	Throughput	%CPU Utilization	Avg Turnaround Time	Avg Waiting Time	% Seek time	% Latency Time
4	99	10.96	1071	1068	38.00	29.64
6	99	9.21	1074	1070	47.90	24.91
8	99	7.94	1077	1073	55.08	21.48
10	99	6.98	1080	1076	60.51	18.88

Table 21. Average Seek Time vs. Performance Measures (SSTF–First Fit)

Avg Seek Time	Throughput	%CPU Utilization	Avg Turnaround Time	Avg Waiting Time	% Seek time	% Latency Time
4	99	16.14	1067	1064	8.69	43.66
6	99	15.47	1068	1064	12.49	41.84
8	99	14.85	1068	1064	15.99	40.17
10	99	14.28	1069	1065	19.22	38.62

Figure 21. Average Seek Time vs. Performance Measures (FIFO–First Fit)

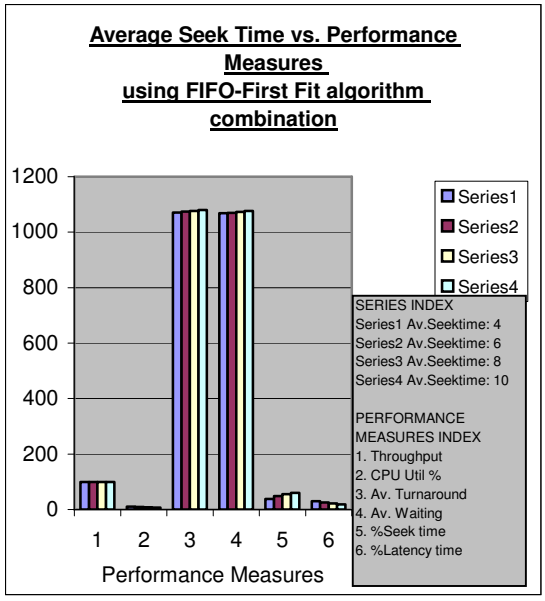
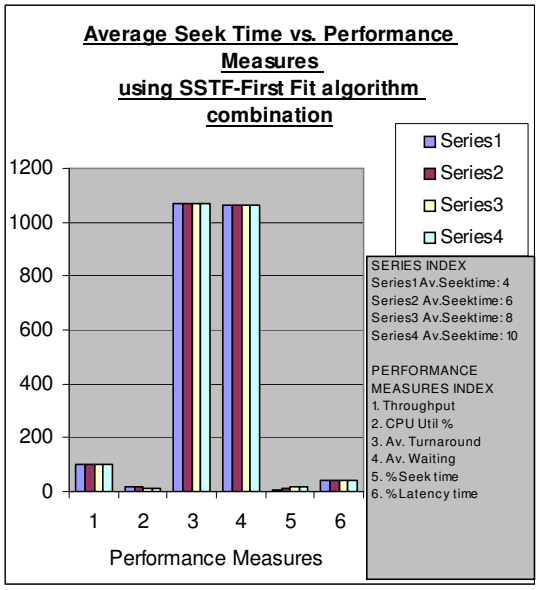


Figure 22. Average Seek Time vs. Performance Measures (SSTF–First Fit)



Average Latency Time vs. Performance Measures

Latency time is objectively measured as average latency time which is time taken for half rotation of the disc platters. Measured in ms, average latency time is derived by the spindle speed. Typical average latency times vary from 2ms to 8ms [7]. For this reason, the average latency time was varied as depicted.

Fixed Parameters:
RR Time Slot: 2ms
Sector Size: 1KB
Simulation Time: 3000ms
Average Seek Time: 8ms

Table 22. Average Latency Time vs. Performance Measures (FIFO–First Fit)

Avg Latency Time	Throughput	%CPU Utilization	Avg Turnaround Time	Avg Waiting Time	% Seek time	% Latency Time
2	99	8.90	1075	1071	61.70	12.03
4	99	7.94	1077	1073	55.08	21.48
6	99	7.17	1079	1075	49.73	29.10
8	99	6.54	1081	1077	45.34	35.36

Table 23. Average Latency Time vs. Performance Measures (SSTF–First Fit)

Avg Latency Time	Throughput	%CPU Utilization	Avg Turnaround Time	Avg Waiting Time	% Seek time	% Latency Time
2	99	18.58	1066	1062	20.01	25.13
4	99	14.85	1068	1064	15.99	40.17
6	99	12.36	1070	1066	13.31	50.18
8	99	10.59	1073	1069	11.41	57.31

Similar to the effect of average seek time discussed earlier, the average latency time tends to affect CPU utilization [see Tables 22 and 23 and Figures 23 and 24]. Higher average latency times relate to higher latency time percentage that pulls down the % CPU utilization. As a special note, the total seek time is unchanged (e.g. in all cases it was 7047ms in case of FIFO-First Fit and 1094 in case of SSTF-First Fit). It is the increasing latency time that pulls down the % seek time. It is important to understand this fact for the associated risk of misinterpreting that higher latency rates improve seek times.

The variable parameter tends to show little if any effect on the other performance measures.

Figure 23. Average Latency Time vs. Performance Measures (FIFO-First Fit)

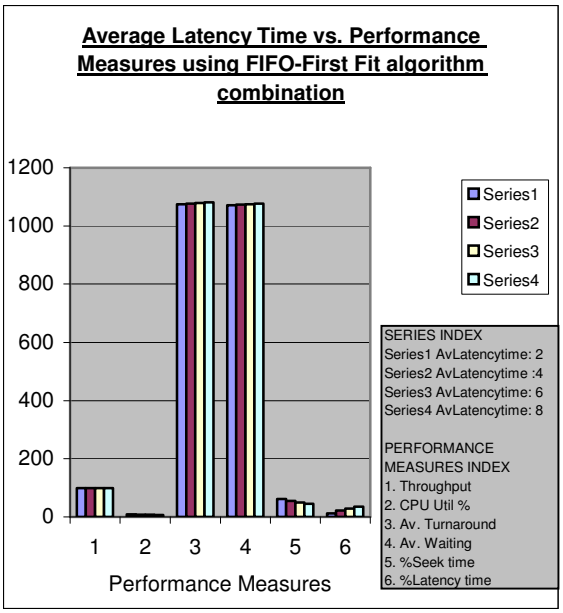
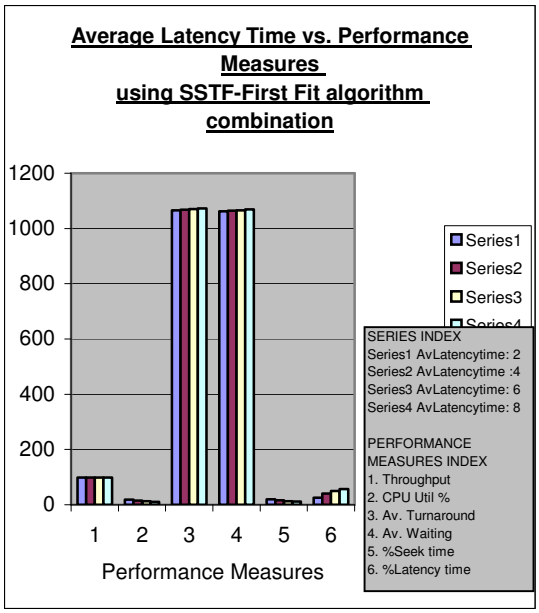


Figure 24. Average Latency Time vs. Performance Measures (SSTF-First Fit)



Round Robin Time Slot vs. Performance Measures

Fixed Parameters:
Average Seek Time: 4ms
Average Seek Time: 8ms
Simulation Time: 3000ms
Sector Size: 1KB

Table 24. Round Robin Time Slot vs. Performance Measures (FIFO-First Fit)

RR Time Slot	Throughput	%CPU Utilization	Avg Turnaround Time	Avg Waiting Time	% Seek time	% Latency Time
2	99	7.94	1077	1073	55.08	21.48
4	168	13.66	719	716	50.89	20.13
6	200	17.32	277	274	47.77	19.77
8	247	20.51	263	259	45.9	19.14

Table 25. Round Robin Time Slot vs. Performance Measures (SSTF-First Fit)

RR Time Slot	Throughput	% CPU Utilization	Avg Turnaround Time	Avg Waiting Time	% Seek time	% Latency Time
2	99	14.85	1068	1064	15.99	40.17
4	168	23.43	714	710	15.73	34.55
6	200	28.39	272	269	14.39	32.41
8	247	34.64	258	254	8.65	32.31

Many interesting trends can be studied here as is evident from the above tables and their corresponding charts [see Tables 24 and 25 and Figures 25 and 26]. Increasing the time slot markedly increases the CPU utilization and throughput, and decreases the average turnaround and average waiting time. All these are indicative of the FIFO behavior. Though performance measures tend to make increased time slot look like a very lucrative proposal, associated disadvantages of possible starvation of processes apply. As the context switch decreases with increasing time quantum, so does the percentage seek and latency times. All this collectively increases the % CPU utilization.

Figure 25. Round Robin Time Slot vs. Performance Measures (FIFO-First Fit)

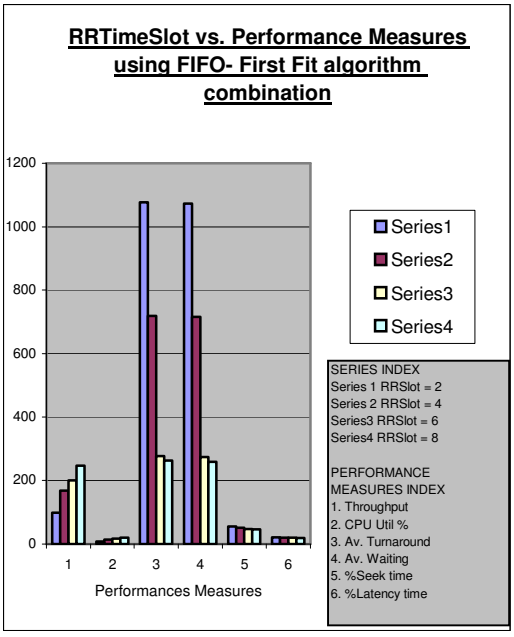
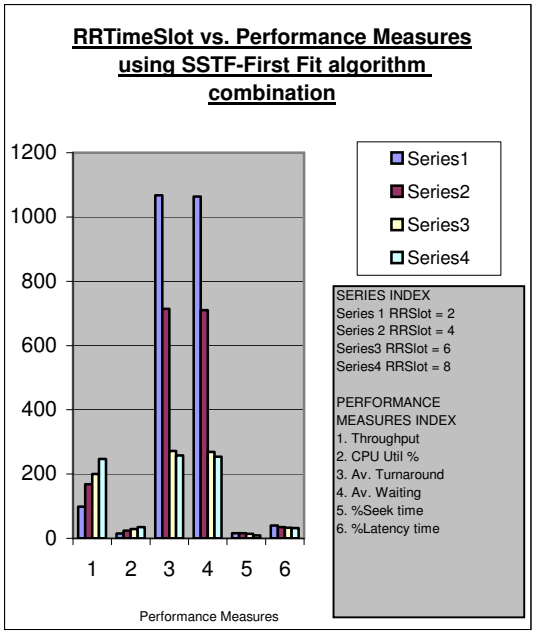


Figure 26. Round Robin Time Slot vs. Performance Measures (SSTF-First Fit)



Sector Size vs. Performance Measures

Fixed Parameters:

Average Seek Time: 4ms Simulation Time: 3000ms

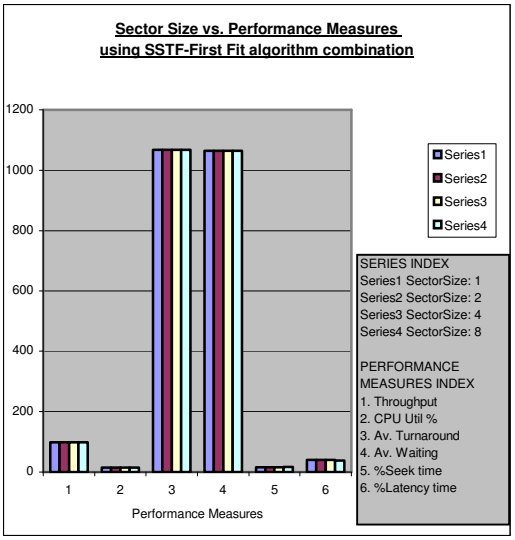
Average Seek Time: 8ms RRTime Slot: 2ms

Table 26. Sector Size vs. Performance Measures (SSTF-First Fit)

Sector Size (KB)	Throughput	%CPU Utilization	Average Turnaround Time	Average Waiting Time	% Seek time	% Latency Time
1	99	14.85	1068	1064	15.99	40.17
2	99	14.85	1068	1064	15.99	40.17
4	99	14.85	1068	1064	15.99	40.17
8	99	15.23	1068	1064	16.44	38.58

This parameter was studied for the SSTF-First Fit algorithm combination and due to lack of interesting results, further data was not collected. The parameter of sector size was varied over the range of 1KB to 8KB and showed little effect on any of the performance measures [see Table 26 and Figure 27]. It can be explained by the fact that sector size does not affect disc allocation markedly and has no say in transfer rate. Moreover, process sizes varied from 20KB to 2MB in multiples of 20KB.

Figure 27. Sector Size vs. Performance Measures (SSTF-First Fit)



To summarize, the SSTF-First Fit algorithm emerges to be the optimal algorithm combination. With the optimal value of round robin time quantum and minimal possible values of average seek time and average latency time, the system performance tends to be maximized. The sector size however, tends to show little if any effect.

3.5. Integration Of The Modules Into An Optimal Operating System

After an in-depth study of the four critical operating systems functions of CPU scheduling, synchronization and deadlock handling, memory management and disc scheduling from a parametric optimization point of view, the next logical step is to integrate the modules to obtain an optimal non-platform based operating system. The results of some such attempts are discussed first.
Identification of the most important parameters is useful for a complete evaluation and for a fair design of a real system. Zhao [6] and Su [4] have done research along these lines i.e. attempting to find the right permutation of design parameters to achieve excellent system performance for various process mixes.

Zhao [6] attempted a simulation of an operating system using multiple level feedback queue for CPU scheduling, paging for memory management and use of FIFO disc scheduling algorithm under disc management. The study identified round robin slot time, aging parameters, page replacement algorithms and page size to be some of the more critical parameters deriving system performance.

Su [4] in her work on parametric optimization of an operating system studied the parameters of CPU scheduling algorithms, prediction of burst time and round robin quantum in context of system performance. The study used Fourier Amplitude Sensitivity Test (FAST) analysis to identify the most important parameters that contribute to system performance [see Appendix A for further details on FAST]. Execution time and number of processes were identified to be the most important factors contributing to the performance of the model system. Round robin time quantum, process size and process priority followed close suit. In terms of parametric optimization, FIFO, priority-base and SJF scheduling fared better as compared to multilevel feedback queue and round robin queue; the impact of round robin quantum was found significant only when the value was small; and performance of SJF and multilevel feedback queues was found to be largely dependent on the prediction of burst time.

Integrated Perspective of our work

The study started with an in-depth discussion of four critical operating system functions namely CPU scheduling, synchronization and deadlock handling, memory management and disc scheduling. We started with CPU scheduling as it is the most elementary and closest to the concept of process and process-mix. Next, we discussed the topic of process synchronization and deadlock handling. As we moved to the memory management module, our simulation integrated CPU scheduling with memory management. The CPU scheduling algorithm chosen, however, was round robin algorithm instead of the multi-level feedback queue. Next, as we moved to disc scheduling, we built on our implementation from memory management module by integrating disc scheduling into the same. In other words, our implementation as discussed under the disc scheduling module can also be viewed as an operating system that uses round robin algorithm for CPU scheduling, continuous memory allocation scheme for memory management and has a disc scheduling mechanism.

The parameters of this integrated system are, hereby, enumerated:

- Time slot for the round robin queue
- Aging time for transitions from Queue 4 to Queue 3, Queue 3 to Queue 2 and Queue 2 to Queue 1 i.e. the aging thresholds for FIFO, priority-based and SJF queues
- α-values and initial execution time estimates for the FIFO, SJF and priority-based queues.
- Choice of preemption for the SJF and Priority based queues.
- Context switching time
- Memory size
- RAM access time
- Compaction algorithm
- Compaction thresholds – Memory hole-size threshold and total hole percentage
- Memory placement algorithms – first-fit, best-fit, worst-fit
- Disc access time (seek time, latency time and transfer time)

- Disc configuration
- Disc scheduling algorithm – FIFO, SSTF, LOOK, C-LOOK, SCAN, C-SCAN
- Disc writing mechanism

Next comes the issue of optimizing the system and coming up with the right permutation of design parameters to achieve excellent performance measures. As was discussed earlier in the paper, even if six of the above mentioned parameters have ten possible values, then we have a million possible permutations. Furthermore, the results obtained from these permutations are applicable to our particular process mix only.

Thus, only the optimal values for the parameters that have been studied as variable independent parameters in the course of this study are enumerated: round robin time – 4ms, α-updating scheme – no effect, memory placement algorithm – best fit, disc scheduling algorithm – SSTF, average seek time – 4ms, average latency time – 2ms, sector size – no effect. The following are the values of the fixed independent variables: RAM size – 32MB, Compaction thresholds – 6% and hole size = 50KB, RAM access time – 14ns, Disc configuration – 8 surfaces, 300 tracks/surface, disc access time – (seek + latency + job size (in bytes)/50000) ms. The above stated optimal values are pertinent to our particular process mix only.

At this point, we would like to reiterate that the purpose of this study has been to present an alternative approach in studying operating systems design. By using parametric optimization, we propose a method of study such that students get a clear concept of operating systems functions and design rather than implementing a real system.

4. Summary and Conclusion

An alternative approach to the study of operating systems design and concepts by way of parametrically optimizing four of the critical operating system functions, namely, CPU scheduling, synchronization and deadlock handling, memory management and disc scheduling has been the focus of this study. Last but not least, an integration of the modules into an optimal operating system with the right permutation of design parameters to achieve excellent performance measures for various process mixes is discussed. A detailed discussion in context of the various modules attempts to make clear the relation of each parameter in the four modules with system performance. While parameters like memory size, transfer rate and number of available resources have a straightforward and well anticipated effect, it is round robin quantum time, memory placement algorithms, disc scheduling algorithms, etc. that tend to be more challenging to optimize.

The goal of presenting parametric optimization of four critical operating system functions in terms of the above stated more challenging parameters has been attempted by means of simulations of the same. The four tasks simulated are CPU scheduling, synchronization and deadlock handling, memory management, and disc scheduling respectively.

It is the optimal value of the round robin quantum along with smallest possible context switching time that tends to maximize performance in context of CPU scheduling. α-updating did not have any effect in our simulations using the specified process-mix.

In the deadlock handling module, the performance measure of rejection rate is controlled by the dynamic mix of number of processes, number of available requests and maximum resource requirement per process and also, by the order in which the requests are made. More available resources and fewer resource requests improve performance in a linear manner. A study of rejection rate over time gave a normal curve.

In context of memory management, using first fit algorithm with a round robin time slot value of two time units constitutes the parametric optimization in terms of the performance parameters including fragmentation time; none of the memory placement algorithms namely first-fit, best-fit and worst-fit emerged as more efficient than the others. The study of fragmentation percentage over time provided probable time windows where compaction could have been undertaken in the course of a simulation.

In the disc-scheduling module, the SSTF-First Fit algorithm combination emerges to be the optimal algorithm combination. With the optimal value of round robin time quantum and minimal possible values of average seek time

and average latency time, the system performance tends to be maximized. The sector size however, tends to show little if any effect.

Presenting the work from an integrated perspective, round robin quantum, aging parameters, page replacement algorithms, page size, execution time, number of processes and process size have emerged to be some of the more important parameters. Last but not least, it needs to be reiterated that the purpose of this study has been to present an alternative approach in studying operating systems design. By using parametric optimization, we propose a method of study such that students get a clear concept of operating systems functions and design rather than implementing a real system.

5. References

1. Batra, P. (2000). Parametric optimization of critical Operating System processes. Bridgeport, CT: University of Bridgeport, Department of Computer Science and Engineering**.**
2. Folk, M. J., Zoellick, B., Riccardi, G. (1998). File Structures: An Object-Oriented Approach with C++. USA: Addison Wesley Inc.
3. Silberschatz, A., Galvin, P.B. (1999). Operating System Concepts (5^{th} ed.). New York: John Wiley & Sons, Inc.
4. Su, N. (1998). Simulations of an Operating System and Major Affecting Factors. Bridgeport, CT: University of Bridgeport, Department of Computer Science and Engineering.
5. Tanenbaum, A.S. (1987). Operating Systems Design and Implementation. New Jersey: Prentice-Hall, Inc.
6. Zhao, W. (1998). Non-Platform Based Operating System Optimization. Bridgeport, CT: University of Bridgeport, Department of Computer Science and Engineering.
7. http://www.storagereview.com/map/lm.cgi/seek

6. Biographies

TAREK M. SOBH received the B.Sc. in Engineering degree with honors in Computer Science and Automatic Control from the Faculty of Engineering, Alexandria University, Egypt in 1988, and M.S. and Ph.D. degrees in Computer and Information Science from the School of Engineering, University of Pennsylvania in 1989 and 1991, respectively. He is currently the Dean of the School of Engineering at the University of Bridgeport, Connecticut; the Founding Director of the Interdisciplinary Robotics, Intelligent Sensing, and Control (RISC) laboratory; a Professor of Computer Science and Computer Engineering; and the Chairman of the Prototyping Technical Committee of the IEEE Robotics and Automation Society.

ABHILASHA TIBREWAL received her B.Sc. and M.Sc. in Home Science with honors in Textile and Clothing from Lady Irwin College, Delhi University, India in 1993 and 1995, respectively, and M.S. in Education and M.S. in Computer Science from University of Bridgeport, CT, USA, in 2000 and 2001 respectively. She is currently employed as Visiting Assistant Professor of Computer Science and Engineering at University of Bridgeport.

7. Appendices

APPENDIX A

Fourier Amplitude Sensitivity Test (FAST)

Function
It is a mathematical technique to identify relative contribution of various parameters in a system to its performance.

Central Concept
For a mathematical model,

$$\mathbf{Y} = \mathbf{F}(\mathbf{X})$$

where,

$\mathbf{Y} = \{y_j;\ j = 1, \ldots, n\}$: model output parameter vector
$\mathbf{X} = \{x_i;\ i = 1, \ldots, m\}$: model input parameter vector
$\mathbf{F}$: function

The ensemble mean of its output y_k is obtained by

$$\langle y_k \rangle = \int \ldots \int y_k(x_1, \ldots, x_m)\, p(x_1, \ldots, x_m)\, dx_1 \ldots dx_m$$

where, 'p' is probability density function for **X**.

The central concept to the FAST method is that the above integral over the multiple input parameter space can be transformed into a one-dimensional integral over a search variable s:

$$\langle y_k \rangle = \lim_{T \to \infty} \frac{1}{2T} \int y_k [x_1(s), \ldots, x_m(s)] \, ds$$

with each input parameter x_i being assigned a frequency $\omega_\omega$using a transformation function.

Advantage

This technique dramatically reduces number of experiments needed to estimate a variation in **Y** in response to various combination of **X**, which may change their values in different probability distributions.

For example, FAST requires only 411 runs for a model with 10 input parameters, each of which has 10 values within a range, while a total of ten billion model runs would be needed with regular approach of sampling technique.