

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

Tarek M. Sobh, Abhilasha Tibrewal

University of Bridgeport

## 1. Introduction

The focus of this paper is to present a new approach to teaching and studying operating systems design and concepts by way of parametrically optimizing critical operating systems functions. By using parametric optimization, the students get an opportunity to build a strong understanding of critical operating systems functions and design without implementing a real system. Moreover, the use of simulation gives them a chance to hone their programming skills and data structure skills as they develop a model of the real system. Specifically, CPU scheduling, memory management, deadlock/synchronization primitives and disc scheduling are the four specific functions studied in the course. This paper presents the general organization of the course with in-depth discussion of two of the critical operating system functions. All the concerned parameters are elaborated upon, focusing on their effect on system performance as well as interaction with other parameters.

## 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 [1]. O’Gorman [2] in his paper provides a comprehensive list of reasons for why operating systems should be part of any computer science curriculum. These include:

- Knowledge of how an operating system does what it does
- Making an informed decision about selection of an operating system
- Improving the performance of an operating system by adjusting values of associated parameters
- Operating systems being the largest pieces of software written provide many ideas and techniques that can be applied in software development

O’Gorman [2], Yun-Lin [3] and Leach [4] in their respective papers have presented the three traditional approaches followed in the study of operating systems. These can be enumerated as:

- a) High level discussion with most programming done in a high level language
- b) General theoretical approach with several real systems added as case studies
- c) Use of emulator programs which emulate special architectures

Krishnamoorthy [5] in his paper describes a course using the first approach. He reports several advantages of involving programming projects such as valuable implementation experience of essential features of an operating system, testing and debugging a large program and validation of principles of operating systems learnt in theory.

Authors of several textbooks including Silberschatz and Galvin [1] and Tanenbaum [6] take the second approach in their presentation of the material. While it is certainly a tried and tested approach, O’Gorman [2] elaborates on potential difficulties in integrating the case studies with the theoretical material.

Case studies of the third approach can be found in the works of Oh and Mossé [7] and Wear et al [8]. The latter enlist direct experimentation with an operating system by varying basic characteristics of the system and the job mix as an effective method to study operating system design. They further state that, “It is prohibitively expensive, difficult and hazardous to allow students to perform such experiments directly on a functional system already allocated for other computing uses”. These factors have been stated as the primary motivation behind designing their operating system simulator by Wear et al [8].

Each of the above methods has its own merits. The approach of parametric optimization presented in this paper provides a healthy balance of the advantages of each method. Moreover, much of the operating-system theory concentrates on the optimal use of computing

## Abstract

Operating systems theory primarily concentrates on the optimal use of computing resources. This paper presents an alternative approach to teaching and studying operating systems design and concepts by way of parametrically optimizing critical operating system functions. Detailed examples of two critical operating systems functions using the presented pedagogical approach are included.

resources. The general outline of the course is presented first followed by two detailed examples of using parametric optimization in CPU scheduling and memory management.

### 3. General Outline of the Course

The course described here is an implementation-oriented course in the structure and design of operating systems. The prerequisites include courses in data structures and computer architecture as well as a good knowledge of C++. The students are free to use other high level languages like Java and C# (C++ is stated as a prerequisite as the introductory programming courses use C++ in the curriculum). The course is open to both undergraduate and graduate students with course deliverables being more rigorous for the graduate audience. The students are proficient in programming by the time they take this course in either case.

The underlying goal of the course is to provide practical experience for the theoretical concepts of the subject. This is done by way of both case studies of real operating systems; and programming projects simulating the concepts and experimenting with them. Furthermore, laboratories assignments on both Windows NT and Unix environments provide breadth to the experience with real operating systems. These include topics such as inter-process communication on the two platforms. The paper does not discuss this part of the course in detail as standard lab manuals are used for this part. It suffices to state that these labs bring in the merits of the second approach (outlined in section 2 of this paper) to the course.

The focus of this paper is the part of the course that integrates the first and third approach (outlined in section 2 of this paper) in form of programming assignments that simulate critical operating system functions and parametrically optimize them.

### 4. Parametric Optimization of Operating Systems Modules

In the next three subsections, the essential components are elaborated upon. Section 4.1 discusses the process control block, Section 4.2 elaborates on the performance parameters and Section 4.3 introduces the different evaluation techniques.

#### 4.1. 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,

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

Table 1. A Sample Process Mix

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) [9]. Figure 1 shows typical process mix and Table 1 illustrates an instance of a process mix.

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 [9].

#### 4.2. Performance Parameters

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

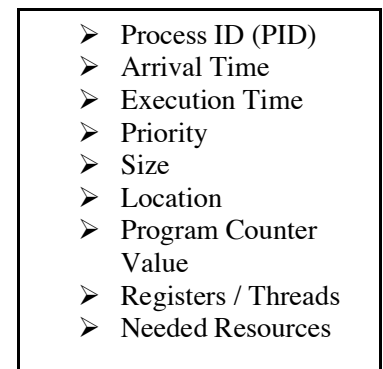


Figure 1. A Typical PCB

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.

### 4.3. 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 [10]. There are several evaluation techniques. Lucas (1971, as cited in [10]) 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 [10]).

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 [10].

The four programming projects in the course simulate and parametrically optimize the tasks of CPU scheduling, synchronization and deadlock handling, memory management and disc scheduling in terms of the involved parameters. The simulation technique is used to analyze some of the stated parameters in their respective modules:

- CPU scheduling: round robin time quantum, aging parameters,  $\alpha$ -values and initial execution time estimates, preemption switches, context switching time.

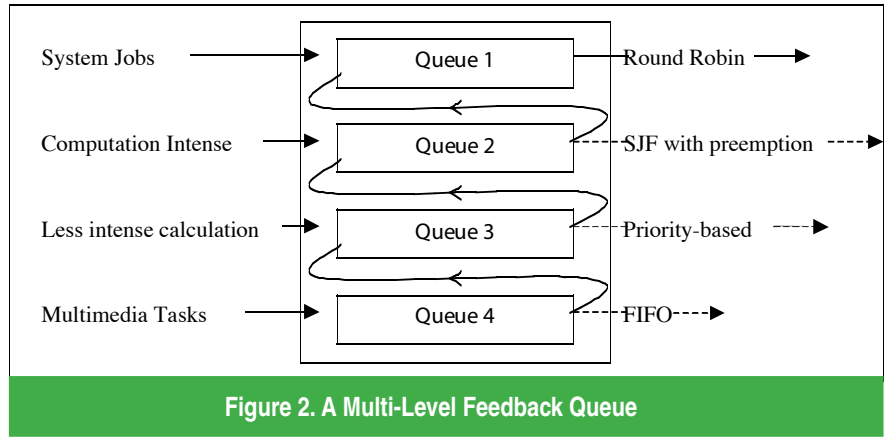
- Synchronization and Deadlock Handling: total number of processes, total number of available resources, maximum number of resources required by the processes, rejection rate over time.
- Memory Management: memory size, RAM and disc access times, compaction thresholds, memory placement algorithms, page 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 is judged by many measures, including: average turnaround time, average waiting time, throughput, CPU utilization, fragmentation, response time, and several other module specific performance measures.

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. 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 the course. Hence, optimization of some parameters in each module is performed to serve as a model example.

The independent variables in the modules 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.

The simulation technique is used to evaluate system performance in all the 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. The parameters



are discussed in terms of their interaction with the operating system function under study and their resultant effect on the system performance. Sub-sections 4.4 and 4.5 present two of the programming projects, CPU scheduling and memory management, in details to exemplify the approach. Section 4.6 discusses the programming projects from an integrated perspective.

#### 4.4. 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.

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.  $\alpha$ , 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

$\alpha$  is the first degree filter and  $0 \leq \alpha \leq 1$

The following example provides a deeper understanding of the issue at hand.

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  $\alpha$  affects the estimation process. Following is the scenario when  $\alpha$  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  $z_{n-1}$

Consequently, a symbolic value of  $\alpha$  is chosen as a starting point to obtain  $f(\alpha)$  i.e. the sum of square difference (see Table 3). Further, differentiation of this and equating it to zero gives the value of  $\alpha$  for which the difference between the actual time and estimated time is minimum. The following exemplifies  $\alpha$ -update in the above example.

In the above example, at the time of estimating execution time of P3,  $\alpha$  is updated as follows.

Processes	$z_n$	$t_n$
P <sub>0</sub>	10	6
P <sub>1</sub>	8	4
P <sub>2</sub>	6	6
P <sub>3</sub>	6	4
P <sub>4</sub>	5	17
P <sub>5</sub>	11	13
P <sub>6</sub>	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  $z_2, z_3, \dots, z_6$  are calculated.

**Table 2. Calculating Execution Time Estimates**

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,

$$8\alpha^3 + 6\alpha^2 + \alpha + 1 = 0 \quad \text{(Equation 1)}$$

Solving Equation 1, one gets  $\alpha = 0.7916$ .

Now,

$$z_3 = \alpha z_2 + (1-\alpha) t_2$$

Substituting values, one gets

$$z_3 = (0.7916) 6 + (1-0.7916) 6$$

$$= 6$$

Next, the parameters involved in a CPU scheduler using the multilevel feedback queue algorithm are discussed.

#### 4.4.1. 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
- $\alpha$ -values and initial execution time estimates for the FIFO, SJF and priority-based

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

**Table 3.  $\alpha$ -updating scheme**

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 [11] 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  $\alpha$ -values and initial execution time estimates: Su [10] 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  $\alpha$  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  $\alpha$  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 [1]. 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  $\alpha$ -updating scheme.

#### 4.4.2. 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  $\alpha$  values for the FIFO, SJF and priority queues are some of the independent variables in this module. To optimize any one of them, every other parameter is kept fixed and the studied parameter varied. Optimization of the round robin time and the effect of the  $\alpha$  update scheme is attempted to serve as a model. Thus, the round robin time was the variable parameter in this 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.

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

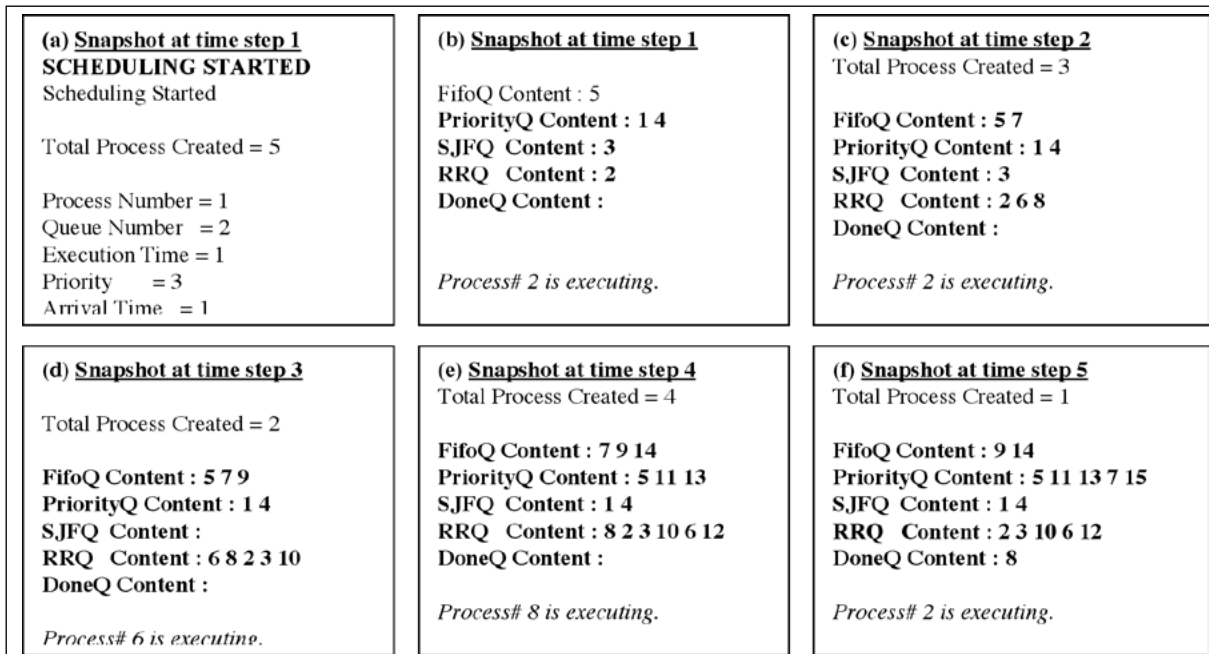


Figure 3. Snapshot of process mix at time steps 1-5

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 this module is included in Appendix A.

#### 4.4.3. 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.

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

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

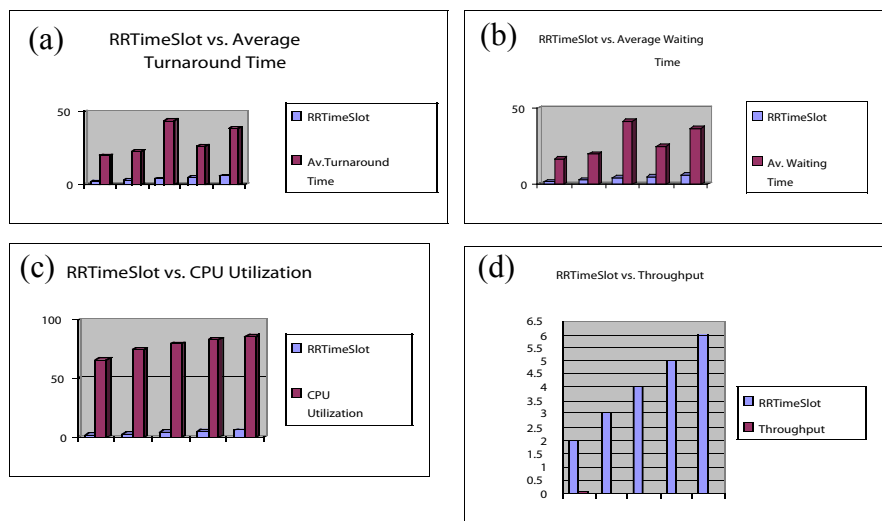


Figure 4. Charts illustrating effect of round robin quantum over performance measures

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.

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

As is evident from Table 5,  $\alpha$  updating did not affect system performance in this case. Again, the result is specific for this 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 this simulation,  $\alpha$ -updating did not tend to affect performance.

#### 4.5. 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

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)

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

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 5. Suppose that the next process requests 18,462 bytes. If exactly the requested block is allocated, one is 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 than the requested memory. The difference between these two numbers is **internal fragmentation** – memory that is internal to a partition, but is not being used [1]. In other words, unused memory within allocated memory is called internal fragmentation [12].

**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 6 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 fragmentation problem can be severe. In the worst case, there could be a block of free (wasted) memory between every two processes. If all this memory were in one big free block, a few more processes could be run. Depending on the total amount of memory storage and the average process size, external fragmentation may be either a minor or major problem.

One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents to place all free memory together in one large block. The simplest compaction algorithm is to move all processes toward one end of the memory, and all holes in the other direction, producing one large hole of available memory. Figure 7 shows different ways to compact memory. Selecting an optimal compaction strategy is quite difficult.

Compaction is an expensive scheme. Given a 128 MB RAM and an access speed of 10ns per byte of RAM, the compaction time becomes twice the product of the two, in this case, 2.56 seconds ( $2 \times 10 \times 10^{-9} \times 128 \times 10^6$ ). Supposing, a round robin scheduling algorithm were used with a time quantum of 2ms, the above compac-

tion time turns out to be equivalent to 1280 time slots.

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 there are two holes of size 'x' and size 'y' respectively and the hole threshold is 4KB, then compaction is done provided  $x \leq 4KB$  and  $y \leq 4KB$ .
- **Total hole percentage:** The total hole per-

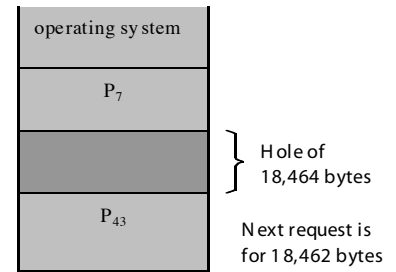


Figure 5. Internal fragmentation

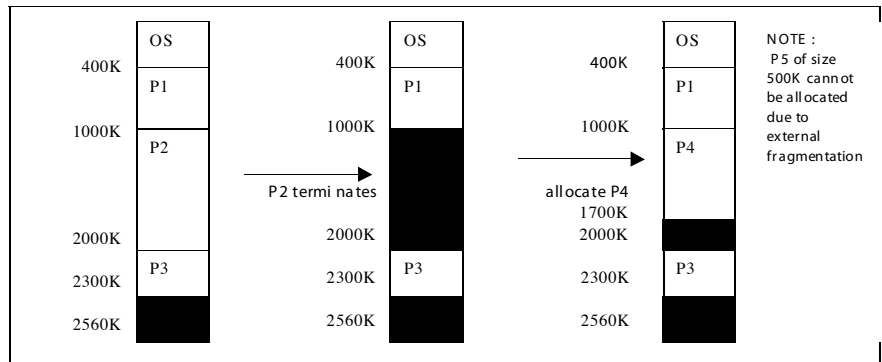


Figure 6. External Fragmentation

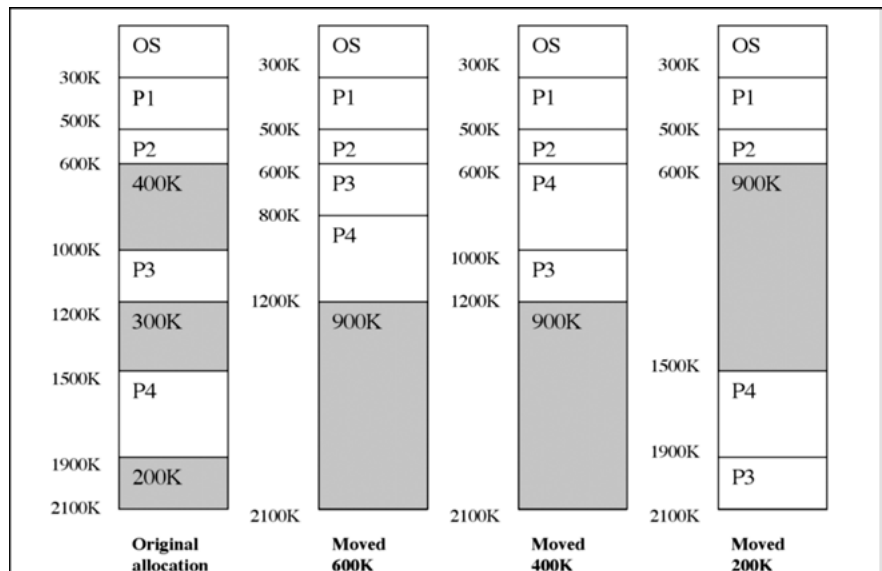


Figure 7. Different ways to compact memory

centage 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  $(x + y) \geq 6\%$  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. Paging is discussed 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 [12].

#### 4.5.1. 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.

##### 4.5.1.1. 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 [1] 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 8). 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 8 is too much in a real system and makes the choice of time slot a difficult proposition to decide upon.

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.

#### 4.5.1.2. 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:

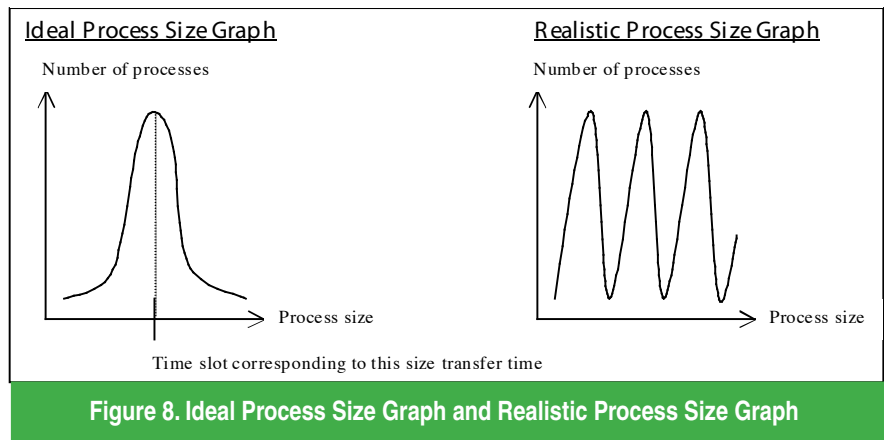
- Memory Size (32 MB)
- Disc access time (1ms (estimate for latency and seek times) + (job size (in bytes)/500000) ms)
- Compaction threshold (6% and hole size = 50KB)
- RAM Access Time (14ns)

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.



**Figure 8. Ideal Process Size Graph and Realistic Process Size Graph**

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 included in Appendix B.

### 4.5.1.3. Simulation Results and Discussion

The round robin time quantum is one of the two variable parameters studied in this simulation.

Table 6 and Figure 9 illustrate the effect of varying the round robin quantum time over the various performance parameters in context of the first fit algorithm.

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, one observes 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 7 and Figure 10 ((a) to (e)).

For this 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 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 than 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

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%

Table 6. Round Robin Time Quantum vs. Performance Measures

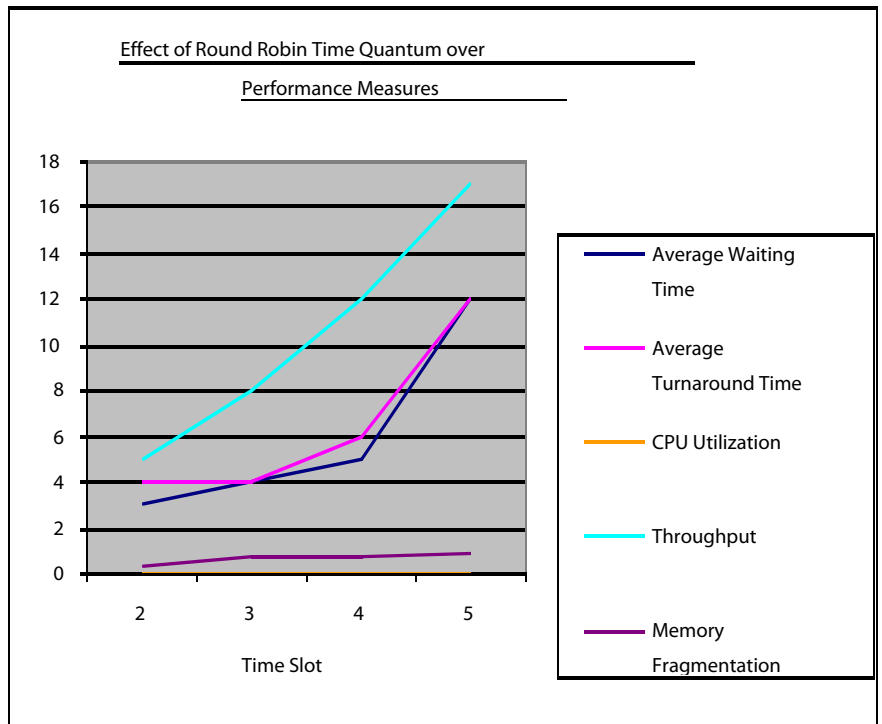


Figure 9. Effect of Round Robin Time Quantum over Performance Measures

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

Table 7. Comparing Memory Placement Algorithms

the time windows. The trend was studied for four different values of round robin time slot. Since the total hole size percentage threshold was specified as 6%, time frames with fragmentation percentage values higher than that were candidates for compaction [see Table 8 and Figure 11]. However, compaction was undertaken in any of the above candidate frames only if the hole size threshold specification was also met.

Looking at Figure 11, one 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.

#### 4.5.2. 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. How-

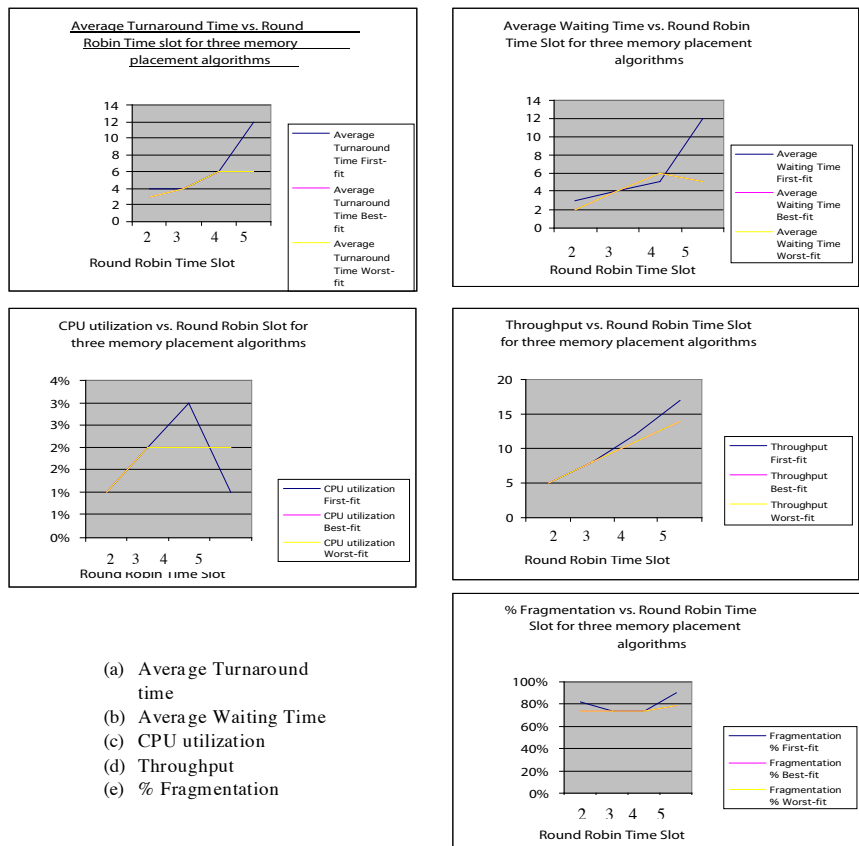


Figure 10. Comparing Memory Placement Algorithms

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

Table 8. Fragmentation percentage over time

ever, 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 memory address. Two of the more significant parameters in a paging scheme are: page size and page replacement algorithms.

Hereby, a paging example with a 64MB RAM and 2KB page size is discussed. 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 12], 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 =  $2^{15} \times 30$  bits  $\approx$  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 =  $2^{16} \times 32$  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.

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 =  $\frac{1}{2} \times$  page size  $\times$  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

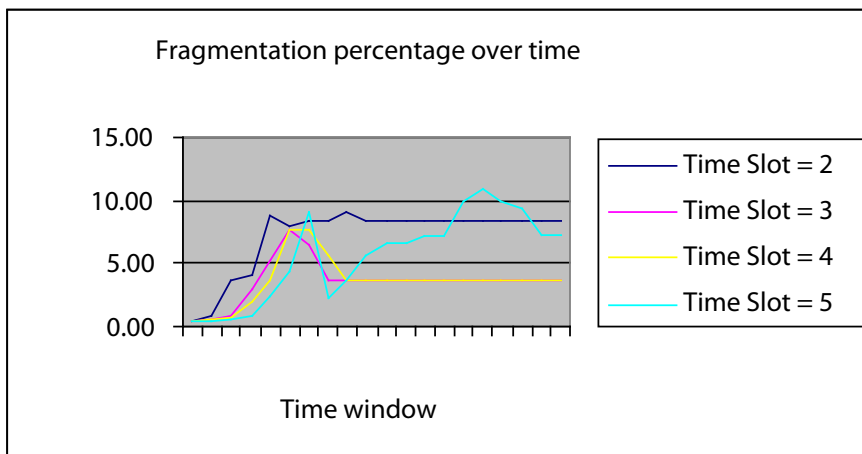


Figure 11. Fragmentation percentage over time

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, multi-level 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.

Figure 12. A Page Table

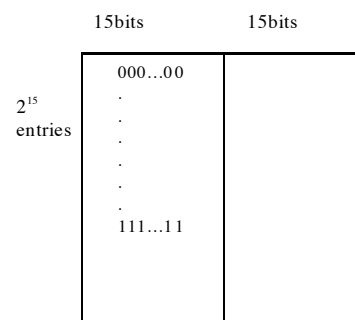


Figure 12. A Page Table

### Continuous Memory Allocation versus Paging Allocation

Table 9 gives a comparison between the two studied memory management schemes.

#### 4.5.2.1. 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 [11] 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.

Continuous Memory Allocation Scheme	Paged Allocation Scheme
<p><u>Advantages:</u></p> <ul style="list-style-type: none"> <li>• An easy algorithm for implementation purposes.</li> </ul>	<p><u>Advantages:</u></p> <ul style="list-style-type: none"> <li>• No external fragmentation, therefore, no compaction scheme is required.</li> </ul>
<p><u>Disadvantages:</u></p> <ul style="list-style-type: none"> <li>• Fragmentation problem makes compaction an inevitable part. Compaction in itself is an expensive proposition in terms of time.</li> </ul>	<p><u>Disadvantages:</u></p> <ul style="list-style-type: none"> <li>• Storage for pagetables.</li> <li>• Addressing a memory location in a paging scheme needs more than one access depending on the levels of paging.</li> </ul>

**Table 9. Comparing continuous memory allocation scheme with paged allocation**

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.

#### 4.5.2.3. Implementation Results

The data in Table 10 (taken from Zhao's study [11]) show the effect of replacement algorithms on the replacement ratio.

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.

#### 4.6. Integrated Perspective

The first programming project in the course starts with CPU scheduling, as it is the most elementary and closest to the concept of process and process-mix. Next, the topic of pro-

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

**Table 10. Page Replacement Scheme vs. Replacement Ratio percentage**

cess synchronization and deadlock handling is undertaken. The class then implements the memory management module, where the simulation integrates CPU scheduling with memory management. The CPU scheduling algorithm chosen, however, is round robin algorithm instead of the multi-level feedback queue. The final programming project is built on the implementation of memory management module by integrating disc scheduling into the same. In other words, the implementation 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
- $\alpha$ -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 a million permutations are possible. Furthermore, the results obtained from these permutations are applicable to one particular process mix only.

Thus, only the optimal values for the parameters that have been studied as variable independent parameters in the individual modules are enumerated. Such a set would include: round robin time – 4ms,  $\alpha$ -updating scheme – no effect, memory placement algorithm – best fit from the two modules presented here as well as optimal values for disc scheduling algorithm, average seek time, average latency time and sector size from the disc scheduling module. The values of the fixed independent variables of the four modules are: 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 a particular process mix only.

## 5. Conclusion

The format of teaching operating systems described in this paper has been followed in the department for several years now. While standard lab exercises enable students to gain experience with Windows NT and UNIX systems, parametric optimization of major operating system functions provide implementation-oriented analytic insight into operating system essentials such as CPU scheduling, deadlock handling, memory management and disc scheduling.

## 6. References

1. Silberschatz, A., Galvin, P.B. (1999). Operating System Concepts (5th ed.). New York: John Wiley & Sons, Inc.
2. O’Gorman, J., Teaching Operating Systems, SIGCSE Bulletin, Vol. 30, June 1998, No. 2, pp. 61-63.
3. Yun-Lin, S., On Teaching Operating Systems, SIGCSE Bulletin, Vol. 21, Sept 1989, No. 3, pp. 11-14.
4. Leach, R., An Advanced Operating Systems Project Using Concurrency, SIGCSE Bulletin, Vol. 22, Sept 1990, No. 3, pp. 39-44.
5. Krishnamoorthy, S., An Experience Teaching Operating Systems Course With A Programming Project, Journal of Computing Sciences in Colleges, Vol. 17, May 2002, No. 6, pp. 25-38.
6. Tanenbaum, A.S. (1987). Operating Systems Design and Implementation. New Jersey: Prentice-Hall, Inc.



7. Oh, J. C. and Mossé, D., Teaching Real Time OSs with DORITOS, SIGCSE Bulletin, Vol. 31, Mar 1999, No. 1, pp. 68-72.
8. Wear, L., Vayda, T. P., MacKenzie, K. and Yakulis, R., An Operating System Model, In Proceedings of the 14th conference on Winter Simulation Vol. 1, San Diego, California, 1982, pp. 323-327.
9. Batra, P. (2000). Parametric optimization of critical Operating System processes. Bridgeport, CT: University of Bridgeport, Department of Computer Science and Engineering.
10. Su, N. (1998). Simulations of an Operating System and Major Affecting Factors. Bridgeport, CT: University of Bridgeport, Department of Computer Science and Engineering.
11. Zhao, W. (1998). Non-Platform Based Operating System Optimization. Bridgeport, CT: University of Bridgeport, Department of Computer Science and Engineering.
12. Folk, M. J., Zoellick, B., Riccardi, G. (1998). File Structures: An Object-Oriented Approach with C++. USA: Addison Wesley Inc.

**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 and Vice Provost for Graduate Studies and Research at the University of Bridgeport, Connecticut; the Founding Director of the Interdisciplinary Robotics, Intelligent Sensing, and Control (RISC) laboratory; and a Professor of Computer Science and Computer Engineering.



**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 Lecturer of Computer Science and Engineering at University of Bridgeport. She is member of ACM, ASEE and the honor societies of Phi Kappa Phi and Upsilon Pi Epsilon.



## 7. Appendices

### APPENDIX A

**NOTE:** Simulation time of 7 has been chosen to show the functioning of aging parameters especially as also other parameters. A simulation time of 7 means that the CPU works for those 7ms and the number of context switches are kept track of separately. This is used when CPU utilization and throughput are calculated where these performance parameters are calculated not on simulation time but simulation time plus total context switches.

#### INITIALIZING SCHEDULER

Please enter the following:

Total simulation time: 7

Maximum allowable processes: 5

Maximum allowable process execution time: 3

Maximum allowable process priority: 5

#### SETTING AGING PARAMETERS

Please enter aging parameters for each Q type:

SJFQ: 1

PriorityQ: 2

FifoQ: 2

#### SETTING PREEMPTION FLAGS

Please set the preemption flags for each Q type:

SJFQ (1/0):0

PriorityQ(1/0):0

#### SETTING ALPHA PARAMETERS

Enter alpha values for the following:

SJFQ:1

PriorityQ:1

FifoQ:1

#### SETTING INITIAL TIME ESTIMATES

Enter execution time estimates for the following:

SJFQ:2

PriorityQ:3

FifoQ:5

#### SETTING QUANTUM TIME FOR ROUNDROBINQ:

Enter quantum time slot for rrq:1

#### SETTING CONTEXT SWITCH:

Enter context switch: 1

#### SETTING MODE:

Step through Each Process or just Output [1/0]:1

#### SCHEDULING STARTED

Scheduling Started

Total Process Created = 5

Process Number = 1

Queue Number = 2

Execution Time = 1

Priority = 3

Arrival Time = 1

Process Number = 2

Queue Number = 4

Execution Time = 3

Priority = 1

Arrival Time = 1

Process Number = 3

Queue Number = 3

Execution Time = 3

Priority = 4

Arrival Time = 1

Process Number = 4

Queue Number = 2

Execution Time = 1

Priority = 3

Arrival Time = 1

Process Number = 5

Queue Number = 1

Execution Time = 3

Priority = 5

Arrival Time = 1

**FifoQ Content : 5**

**PriorityQ Content : 1 4**

**SJFQ Content : 3**

**RRQ Content : 2**

**DoneQ Content :**

*Process# 2 is executing.*

Total Process Created = 3

Process Number = 6

Queue Number = 4

Execution Time = 3

Priority = 1

Arrival Time = 2

Process Number = 7

Queue Number = 1

Execution Time = 3

Priority = 4

Arrival Time = 2

Process Number = 8

Queue Number = 4

Execution Time = 1

Priority = 1

Arrival Time = 2

**FifoQ Content : 5 7**

**PriorityQ Content : 1 4**

**SJFQ Content : 3**

**RRQ Content : 2 6 8**

**DoneQ Content :**

*Process# 2 is executing.*

Total Process Created = 2

Process Number = 9

Queue Number = 1

Execution Time = 2

Priority = 3

Arrival Time = 3

Process Number = 10

Queue Number = 4

Execution Time = 2

Priority = 4

Arrival Time = 3

**FifoQ Content : 5 7 9**

**PriorityQ Content : 1 4**

**SJFQ Content :**

**RRQ Content : 6 8 2 3 10**

**DoneQ Content :**

*Process# 6 is executing.*

Total Process Created = 4

Process Number = 11

Queue Number = 2

Execution Time = 2

Priority = 2

Arrival Time = 4

Process Number = 12

Queue Number = 4

Execution Time = 1

Priority = 1

Arrival Time = 4

Process Number = 13

Queue Number = 2

Execution Time = 1

Priority = 2

Arrival Time = 4

Process Number = 14

Queue Number = 1

Execution Time = 1

Priority = 4

Arrival Time = 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.*

Total Process Created = 1  
Process Number = 15  
Queue Number = 2  
Execution Time = 2  
Priority = 2  
Arrival Time = 5

**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.*

Total Process Created = 4  
Process Number = 16  
Queue Number = 2  
Execution Time = 2  
Priority = 1  
Arrival Time = 6

Process Number = 17  
Queue Number = 2  
Execution Time = 1  
Priority = 4  
Arrival Time = 6

Process Number = 18  
Queue Number = 4  
Execution Time = 1  
Priority = 4  
Arrival Time = 6

Process Number = 19  
Queue Number = 1  
Execution Time = 3  
Priority = 2  
Arrival Time = 6

**FifoQ Content : 14 19**  
**PriorityQ Content : 5 11 13 7 15 9 16 17**  
**SJFQ Content : 1 4**  
**RRQ Content : 3 10 6 12 18**  
**DoneQ Content : 8 2**  
*Process# 3 is executing.*

Total Process Created = 0  
**FifoQ Content : 19**  
**PriorityQ Content : 5 11 13 7 15 9 16 17 14**  
**SJFQ Content : 1 4**  
**RRQ Content : 10 6 12 18 3**  
**DoneQ Content : 8 2**  
*Process# 10 is executing.*

#### **SCHEDULING FINISHED**

**Total Processes Created in the system: 19**  
**Total Processes Finished Execution in system: 2**

**Total Context Switches: 5**

**Maximum TurnAround Time in the system: 5**

**Maximum Waiting Time in the system: 2**

**TotalWaitingTime:4**

**Average Waiting Time in the system: 2**

**TotalTurnaroundTime:8**

**Average TurnAround Time in the system: 4**

**CPU Throughput for this sample run: 0.1667**

**CPU Utilization for this sample run: 58.33%**

**Number of Processes Executed from Round**

**Robin Queue: 2**

**Number of Processes Executed from Shortest**

**Job Queue: 0**

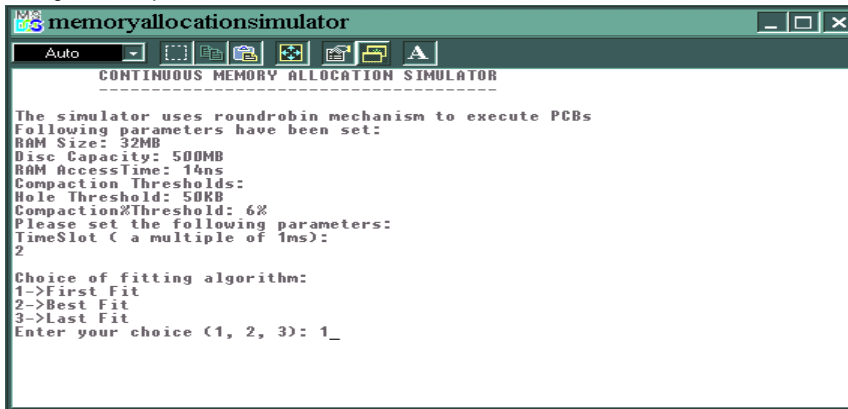
**Number of Processes Executed from Priority**

**Queue: 0**

## APPENDIX B

Walk through a sample run of memory manager module simulation

### 1. Setting variable parameters

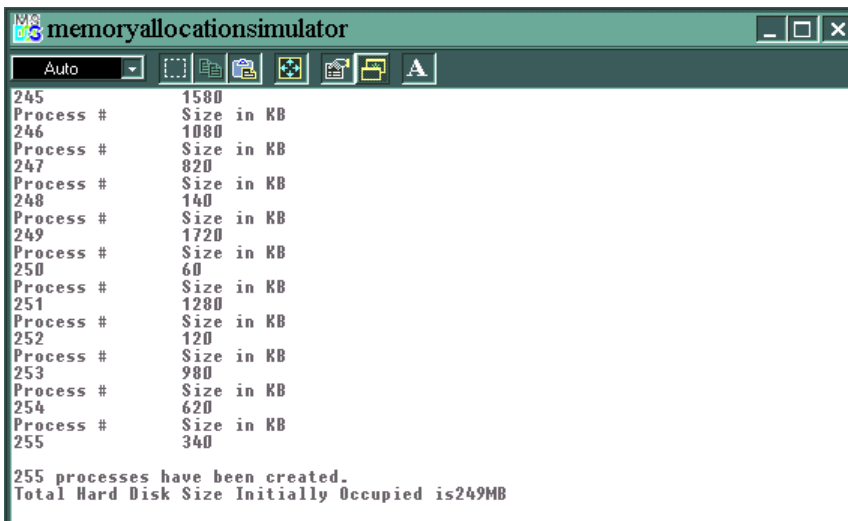


```
memoryallocationsimulator
Auto
CONTINUOUS MEMORY ALLOCATION SIMULATOR

The simulator uses roundrobin mechanism to execute PCBs
Following parameters have been set:
RAM Size: 32MB
Disc Capacity: 500MB
RAM AccessTime: 14ns
Compaction Thresholds:
Hole Threshold: 50KB
CompactionThreshold: 6%
Please set the following parameters:
TimeSlot ( a multiple of 1ms):
2

Choice of fitting algorithm:
1->First Fit
2->Best Fit
3->Last Fit
Enter your choice (1, 2, 3): 1_
```

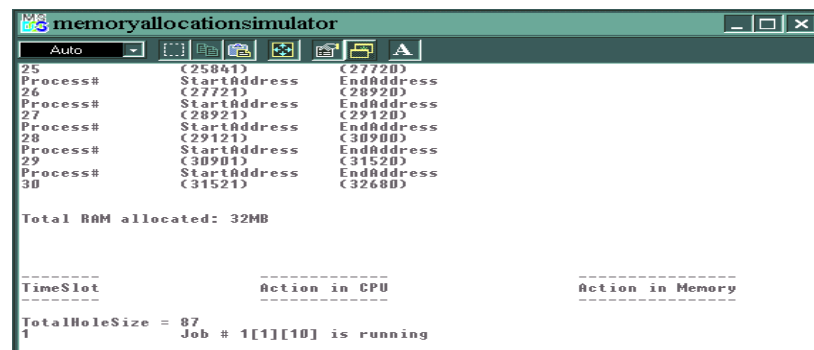
### 2. Initial Hard Disc Configuration



```
memoryallocationsimulator
Auto
245      1580
Process # Size in KB
246      1080
Process # Size in KB
247      820
Process # Size in KB
248      140
Process # Size in KB
249      1720
Process # Size in KB
250      60
Process # Size in KB
251      1280
Process # Size in KB
252      120
Process # Size in KB
253      980
Process # Size in KB
254      620
Process # Size in KB
255      340

255 processes have been created.
Total Hard Disk Size Initially Occupied is249MB
```

### 3. Initial RAM Configuration

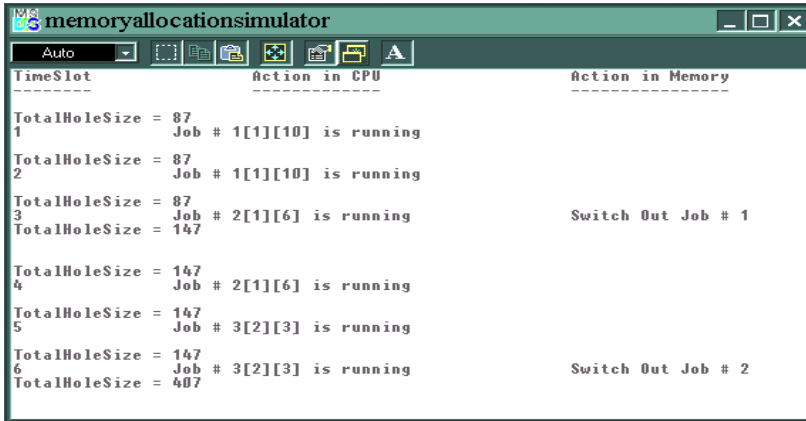


```
memoryallocationsimulator
Auto
25      (25841)      (27720)
Process# StartAddress EndAddress
26      (27721)      (28920)
Process# StartAddress EndAddress
27      (28921)      (29120)
Process# StartAddress EndAddress
28      (29121)      (30900)
Process# StartAddress EndAddress
29      (30901)      (31520)
Process# StartAddress EndAddress
30      (31521)      (32680)

Total RAM allocated: 32MB

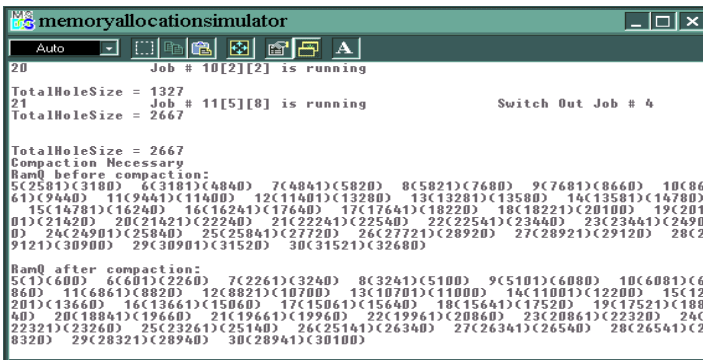
-----
TimeSlot      Action in CPU      Action in Memory
-----
TotalHoleSize = 87
1      Job # 1[1][10] is running
```

4. In the midst of execution. The first column shows simulated time instance, the second one shows the action in the CPU at that instance and the third one shows the action in the Memory at that instance. In addition the total hole size is output at each instance.



5. Compaction Scenario → The first set shows the processes in the RAM prior to compaction and the second one shows the processes in the RAM after compaction. The format is: Process number (starting address) (end address).

Note that after compaction the first process has a starting address of one and each subsequent process has a starting address consecutive to the previous process's end address. In other words, all the holes are compacted to a large one at the end of the RAM.



6. Final Performance Measures For The Run

