FAIRNESS IN PROCESSOR SCHEDULING IN TIME SHARING SYSTEMS

S.Haldar and D.K.Subramanian
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012
India

ABSTRACT: Loosely, fairness is the assurance of granting each request, from a set of requests, within a predetermined bounded time even though the request scheduling primitives are unfair or random. The fairness property has been studied, in solutions to the mutual exclusion problem, from theoretical point of view. Many fair solutions to the mutual exclusion problem have been proposed in recent years. This paper presents one illustration to show how the concepts of fairness could be incorporated in processor scheduling in time sharing systems. First, it looks at some shortcomings associated with the round robin scheduler used in the time sharing systems, and then presents a fair version of the round robin scheduler.

I. INTRODUCTION

In a multi-user computer system, many user processes are executed concurrently to improve the performance of the system, to improve the utilization of the system-resources like processor, disk, printer. The processes use the resources to accomplish some tasks, leading to competition for the resources. If the resources are not scheduled (assigned to the processes) in a fair manner, the system performance as well as the performance of the individual process(es) might degrade considerably.

The assignment of resources to the processes may be thought of a variant of the mutual exclusion problem. At a time, a resource might be used by a single process, i.e., the use of a resource is mutually exclusive. The resource assignment policy must not lead to starvation; moreover it should be fair to each competing process. A solution to the resource assignment problem is said to have k-fair (i.e., k-bounded waiting) property if once a process **p** requested a resource **r** and another process **q**, which has requested **r** later than **p**, could not use **r** more than **k** times ahead of **p** [5, 15] (assuming a process releases a resource **r** after using it. The process might again request for **r**, release it, and so on). Many fair solutions to this type of assignment problem have been proposed in the literature. Among them, the 1-fair solutions proposed in [7, 8] are interesting. Briefly, the 1-fair solution of [8] is the following. Concurrent requests for a particular resource **r** are partitioned into two groups: current group and waiting group. Any new request for r is put in the waiting group. The requests in the current group are served one by another in a random manner. When the current group becomes empty, the roles of the groups are interchanged. Thus the solution ensure 1-fair property even the scheduling of request is unfair. The fairness concept of this solution could be tactfully incorporated in scheduling of resources to improve the system performance. This is the main objective of this paper.

First, the scheduling of processor to the competing processes in time sharing systems is studied in Section II. This section first points out some demerits of conventional round robin scheduling used in time sharing systems, and then presents a fair version of the round robin scheduler. Section III summarizes the results.

II. A VIRTUAL ROUND ROBIN SCHEDULER

II.1 Introduction

Round robin scheduler is widely used in general purpose time sharing system to assign processor to the competing processes. The system ensures faster response time to shorter processes. The working principle of the system is the following [14]. A process enters the system by joining a *"first in first out"* queue called ***ready queue.*** Each time the process reaches the head of the ready queue, it runs on the processor for no longer than a predetermined time interval called a ***quantum.*** If the process is not complete in the current quantum, it releases the processor and joins the end of the ready queue for next selection. If the process initiates an input/output (I/O) activity, it releases the processor and joins an appropriate device queue. The process again joins at the end of the ready queue as soon as its I/O is done. In rest of the section we call it conventional round robin (**CRR**, in short) scheduler.

The main advantages of **CRR** scheduler are its simplicity and its low decision making overhead. It is one of the best known schedulers for achieving good and relatively evenly distributed terminal response time. However, a few anomalies exist in a system that uses **CRR** scheduler. A system using **CRR** scheduler behaves uniformly and evenly with all the competing processes provided the processes are alike, i.e., their execution patterns are almost same. For example, if all the competing processes are completely CPU-bound, the system behaves fairly with each individual process. ***(A process is regarded as CPU-bound if it mainly performs computational work and occasionally uses I/O devices. Contrarily, a process is I/O-bound if it spends more time in using I/O devices than the CPU. Generally an I/O-bound***

process has a shorter CPU burst than that of a CPU-bound process). It is to be noted that it is preferable to have a balanced mix of CPU-bound and I/O-bound processes in the system to optimize the utilization of both the processor and I/O devices. Unfortunately, in such a mixed environment the system does not behave uniformly, evenly and fairly with each individual process. In such a mixed application environment, an I/O-bound process uses a processor for a short period, releases the processor, waits for I/O to be completed, again joins the end of the ready queue, and waits for the next selection. On the contrary, a CPU-bound process uses a complete time quantum at every selection. That is, CPU-bound processes monopolize the processor and I/O-bound processes receive unfair treatment. This results in poor performance of I/O-bound processes, poor utilization of I/O devices, and increase in the variance of response time.

A large number of CPU-schedulers have been developed in the early era of time sharing systems (e.g., [1, 2, 3, 6, 9, 10, 12, 13, 17, etc.]). Unfortunately, each of them concentrated on the performance improvement for a specific category of processes. The performance improvement of one category of processes can only be done at the cost of the processes in other categories. Scheduling algorithms differ only in their choices of the processes to be given preferential treatment. A number of variants of **CRR** scheduler are summarized in [19, p237].

It is observed (in [4, 16, 18, etc.]) that in a computer system having mixed CPU-bound and I/O-bound processes, if CPU-bound processes are given preferential treatment, the performance of I/O-bound processes degrades considerably; on the contrary, if I/O-bound processes are given higher priority, the performance of CPU-bound processes are at stake. The only way to have better utilization of I/O devices is to give higher priority to the I/O-bound processes. It has also been reported in the literature that if I/O-bound processes are given preferential treatment, greedy and clever users incorporate dummy I/O instructions in their programs to get faster response, resulting in a poorer response to genuine users.

Now, consider another objective of relevance to scheduling algorithms. Peterson and Silberschatz state (p104, [14]) - 'It has also been suggested that for interactive systems (such as time-sharing systems), it is more important to minimize variance in the response time than to minimize the average response time. A system with reasonable and predictable response time may be considered better than a system which is faster on the average, but highly variable. There has been very little work done on CPU-scheduling algorithms to minimize variance'.

This paper has taken the above factors into consideration, and suggests an excellent modification of the **CRR** scheduler.

II.2 The Philosophy of the Modified Scheduler

This section presents some overview about the modified scheduler called **VRR** scheduler. The design of the **VRR** scheduler is based on the working principle of **CRR** scheduler and the concept of ***dynamically shrinking time quanta.*** The **VRR** scheduler retains the advantages of a **CRR** scheduler, and at the same time improves (tries to improve) the performance of I/O-bound processes to the level of the performance of CPU-bound processes in a mixed environment. A system using the **VRR** scheduler behaves (tries to behave) uniformly, evenly and fairly with each process. It treats all the processes equally, but some are treated more equally.

Briefly, the working principle of a time sharing system that uses the **VRR** scheduler is as follows. A process enters the system by joining at the end of a *"first in first out"* queue called ***main queue.*** Each time the process reaches the head of the main queue, it runs on a processor for no longer than a predetermined time interval called a ***quantum.*** If the process is not complete in the current selection, it releases the processor, joins the end of the main queue, and waits there for next selection. If the process initiates an I/O activity, it release the processor, and waits in an appropriate device queue for its I/O completion. Then the process joins at the end of another *"first in first out"* queue called ***auxiliary queue*** as soon as its I/O is done. Processes in the auxiliary queue get (non-pre-emptive) higher priority than those in the main queue. When the process reaches the head of the auxiliary queue, it runs on a processor no more than a slice of a **quantum** minus total time spent on processor(s) from its last selection from the main queue.

The **VRR** scheduler incorporates the concepts of ***"1-bounded waiting"*** and ***"dynamically shrinking time quanta"*** in **CRR** scheduler. With the introduction of "1-bounded waiting, the **VRR** scheduler achieves fairness property in assigning the processor to the competing processes. Moreover, the scheduler minimizes the variance of response time when there is a mix of CPU-bound and I/O-bound processes in the system.

II.3 Various Events and Actions

The architecture of the system considered for the present study is shown in the Figure 1, and the queuing model of the same system is shown in Figure 2. The activities of the system are driven by a number of events. Before describing the events and the corresponding actions taken by the system, a few definitions are given next.

II.3.1 Definitions

Remaining Time Slice

The concept of ***remaining time slice*** comes from the concept of "1-bounded waiting" and "dynamically shrinking time quanta". Here the concept of "1-bounded waiting" is formulated for real time. Let a group of processes, viz., $p_1, p_2, \ldots, p_n$, be waiting for a preemptable shared resource **r**. A process p_i in the group is not allowed to use the resource **r** for more than one complete quantum of **Q** time units by any means before each of the other processes in the group has at least once used **r** (may be for a fraction of a quantum). Let process p_i be currently using **r**. Now, if process p_i releases **r** at time Q', where $Q' < Q$, and again becomes interested in using **r**, the process *may be* allowed to use r (a preferential treatment) for a slice of **(Q - Q')** time units before the other processes in the group could use **r**. Process p_i could perform a sequence of release and re-acquisition of r till **(Q - Q')** shrinks to zero. As soon as **(Q - Q')** becomes zero, the process does not get any preferential treatment. If the process is still interested in using **r**, it could use it only when the other processes in the group have completed the procedure mentioned above. That is, when a process starts waiting for a resource, another process cannot use the resource more than one time quantum even if it is given temporary preferential treatment. This retains the property of "1-bounded waiting".

The ***remaining time slice***, t_i of a process p_i, is defined as a slice of $(Q - (Q' \bmod Q))$ time units, where Q be the system defined time quantum and Q' be the total processor time used by the process.

Assumption: Generally, the operating system of a computer stores the *pros and cons* of each process in a separate process control block. It is assumed that the system could save and restore the value of *remaining time slice,* the value of a timer, to and from the respective process control blocks at the time of context switching.

Process States and Transition: Depending on its activity, a process may be in one of the four states - ***ready, running, waiting and terminated.*** Figure 3 shows the state transition model of a process. Consider a typical process as it moves through the state transition model. Initially, the process is in ***terminated*** state. When the process is initiated for some computational work, it is put in a queue of ready processes, and it is ***ready*** to run as soon as the system schedules it on a processor. The processor scheduler will eventually pick the process to execute, and the process enters the state ***running.*** After a period of time, a system clock (or timer) may interrupt the processor on which the process is running, leading to the execution of clock interrupt handler routine. When the clock interrupt handler routine finishes servicing the clock interrupt, the system may decide to schedule another process to execute, so the first process enters the state ***ready*** to run. When the process running on a processor, requires an I/O from or to a device, the process must wait for the I/O to complete and it enters the state ***waiting.*** When the I/O later is done, the hardware interrupt the processor, and the corresponding device interrupt handler awakens the process causing it to enter the state ***ready*** to run. When the process completes its work, it invokes an exit system call and thus enters the state ***terminated.***

II.3.2 Events and Actions

It is assumed that initially all the processes are waiting at their respective terminals for inputs.

Initiation of a Process

When a process finishes its terminal read, it leaves the terminal where it was waiting, changes its state to ***ready*** and joins at the end of the ***main queue*** to be selected by the scheduler for execution.

Exit of a Running Process

When a ***running*** process completes its execution, it releases the processor on which it was running, and goes back to the terminal where it was originally waiting. The process changes its state to ***waiting***, and activates the scheduler.

I/O System Call

When a process running on the processor initiates a device I/O operation, at the first the context of the process (including the value of the timer associated with the processor) is saved in its process control block. Then the process changes its state to ***waiting***, joins the requested device queue, initiates device I/O operation, and activates the scheduler. The process remains in the device queue till its I/O is done.

I/O Device Interrupt

Upon interrupted by an I/O device, the processor puts the process that is doing I/O at the end of the ***auxiliary queue.*** The process state is changed to ***ready.*** The process waits in the auxiliary queue for next selection.

Timer Interrupt

A processor upon interrupted by its timer suspends the execution of its currently ***running*** process, saves the context of the process, and puts the process at the end of the main queue. The process state is changed to ***ready***. Then the scheduler routine is executed.

Activation of the Scheduler

On each of its activation, the virtual round robin scheduler selects a ready process for execution. The selection routine is as follows.

```
If (the auxiliary queue is not empty) then
    select a process from the auxiliary-queue head;
    change the state of the process to running;
    load the context of the selected process and
    set the timer equal to the remaining time slice of the process;
    /*  transfer control to the selected process */
else  if (the main queue is not empty) then
          select a process from the main-queue head;
          change the state of the process to running;
          load the context of the selected process and
          set the timer equal to the system time quantum;
          /*  transfer control to the selected process */
      else
          run (an idling process)
      endif
 endif;
```

II.4 Simulation Study

To evaluate the performance, and the effectiveness, of the **VRR** scheduler, over **CRR** scheduler, an extensive simulation study is done. Some of the results are presented here.

The model chosen here is a central server model of [11]. The time sharing system simulated here consists of a single processor (N.CPU = 1) connected with infinite primary memory and four disks (N.DISK = 4). All the disks are of same characteristics: maximum seek time is 55 msecs, maximum latency time is 20 msecs, and data transfer time is 0.716 msec. Disk service discipline is FIFO, and service time is uniformly distributed between 0.716 and 75.716. There are N.TER (variable parameter) terminals connected to the system. Users initiate processes by submitting commands from the terminals. A terminal, having submitted a command and not yet received any response, are blocked from sending further commands. Command submission rate from a free terminal is a Poisson process with mean λ. The value of $1/\lambda$ is assumed to be 1.2 secs. It is assumed that command-programs are memory resident. Command execution time is considered to be independent, identical random variable distributed exponentially with mean $1/\mu$. The value of $1/\mu$ is assumed to be 1.6 secs. Processes (commands) are classified into two categories: I/O-bound and CPU-bound. An I/O-bound process performs one disk I/O in every $1/\lambda_{i/o}$ time units. and a CPU-bound process performs one disk I/O in $1/\lambda_{cpu}$ time units. The values of $1/\lambda_{i/o}$ and $1/\lambda_{cpu}$ are assumed to be 0.4 sec and 1.6 secs, respectively, i.e., on the average I/O-bound processes do four times disk I/O than CPU-bound processes. The value of time quantum, Q, is assumed to be a constant of 400 msecs.

To avoid interpreting voluminous simulation data, single working environment, (an environment that is characterized by a balanced mix of CPU-bound and I/O-bound processes, i.e., half of the users submit I/O-bound commands and the rest half submit CPU-bound commands), are presented here. Intuitively, it may be said that the performance of the system for pure workload environment, either 100% CPU-bound processes case or 100% I/O-bound processes case, will be the same irrespective of scheduler, **CRR** scheduler or **VRR** scheduler, used because the functions of these schedulers are similar in a pure workload environment. The results obtained from various simulation runs are presented in Figures 4-10.

In the rest of the text, by average of a performance parameter of CPU-bound (or I/O-bound) processes, it is meant that the average is calculated by considering the output of only CPU-bound (or I/O-bound) processes. The system average value of an output parameter is calculated by considering the simulation output of all processes.

Average Response Time

Figure 4 shows how the average response time increases with the increase in number of terminals (N.TER) for the assumed system. It is seen from the figure that the average response time of I/O-bound processes increases rapidly for the system that uses **CRR** scheduler (**CRRS**, in short). While in the case of CPU-bound processes it increases very slowly. In **CRRS**, the average response time of I/O-bound and CPU-bound processes increases form 4.91 secs and 2.89 secs respectively at N.TER = 2, to 332.2 secs and 160.2 secs respectively at N.TER = 100. That is, at N.TER = 100, the average response time of I/O-bound processes is double than that of CPU-bound processes. A considerable improvement of response time characteristic is observed when the same system is brought under the control of the **VRR** scheduler (**VRRS**, in short). In **VRRS**, the average response time of I/O-bound and CPU-bound processes increases form 4.87 secs and 2.65 secs respectively at N.TER = 2, to 251 secs and 188 secs respectively at N.TER = 100. At N.TER = 100, the average response time of I/O-bound processes is only 1.335 times that of CPU-bound processes. At N.TER = 100, there is a reduction in average response time of I/O-bound processes by 24.4%, and an increase in average response time of CPU-bound processes by 17.35% for **VRRS** in comparison to **CRRS**. The system average response time characteristics for both the systems, **CRRS** and **VRRS**, are almost overlapped. The system average response time for **CRRS** and **VRRS** increases from 3.7 secs and 3.525 secs respectively at N.TER = 2, to 216.8 and 215 secs respectively at N.TER = 100.

Let us look at the ratio of average response times of I/O-bound and CPU-bound processes plotted in Figure 5. In the **CRRS**, the ratio increases from 1.7 at N.TER = 2, to 2.0 at N.TER = 20, and remains approximately at the value of 2.0 beyond 20 terminals. Whereas, in the **VRRS**, the ratio decreases form 1.58 at N.TER = 2, to 1.36 at N.TER = 20 and remains closer to 1.36 beyond 20 terminals. From the ratio characteristics it can be concluded that **VRRS** behaves fairly with each process.

Standard Deviation of Response Time

Standard deviation of response time is an important performance index. It is already mentioned in Section II.1 that it is more important to minimize variance (square of standard deviation) of response time than to minimize average response time; a system with reasonable and predictable response time may be considered better than a system which is faster on the average, but highly variable. Figure 6 shows how the standard deviation of response time increases with the increase in number of terminals (N.TER) for the assumed system. In **CRRS**, the standard deviation of response time of I/O-bound and CPU-bound processes increases form 4.92 secs and 2.86 secs respectively at N.TER = 2, to 337.2 secs and 140.5 secs respectively at N.TER = 100. At N.TER = 100, the ratio of standard deviations of response times of I/O-bound and CPU-bound processes is 2.40. Although the average execution time of each process is the same, the standard deviation of response time of I/O-bound processes is 140% higher than that of CPU-bound processes. A considerable improvement in standard deviation of response time characteristics is observed when the system is **VRRS**. In **VRRS**, the standard deviation of response time of I/O-bound and CPU-bound processes increases form 4.946 secs and 2.57 secs respectively at N.TER = 2, to 228.4 secs and 165.7 secs respectively at N.TER = 100. At N.TER = 100, the ratio of standard deviations of response times of I/O-bound and CPU-bound processes is 1.38. The standard deviation of response time of I/O-bound processes is only 38% higher than that of CPU-bound processes in **VRRS**. The characteristic of system standard deviation of response time for **VRRS** is better than that for **CRRS**. The system standard deviation of response time for **CRRS** and **VRRS** increases from 3.96 secs and 3.85 secs respectively at N.TER = 2, to 239 secs and 198 secs respectively at N.TER = 100. Now, consider the point N.TER = 80 for performance comparison. At the point, the standard deviation of response times of I/O-bound processes for **CRRS** and **VRRS** are 267 secs and 184 secs respectively, i.e., there is a reduction in standard deviation by 31.1% in **VRRS**. At N.TER = 80, the standard deviation of response times of CPU-bound processes for **CRRS** and **VRRS** are 110 secs and 131 secs respectively, i.e., there is an increase in standard deviation by 19.1% in **VRRS**. Let us look at the ratio of standard deviation of response times of I/O-bound and CPU-bound processes plotted in Figure 7. In **CRRS**, the ratio values are closer to 2.3. Whereas, in **VRRS**, the ratio values are closer to 1.4. From the Figures 6 and 7 it can be said that **VRRS** behaves (tries to behave) evenly and fairly , if not equally, with each type of process, and **CRRS** behaves non-uniformly and erratically with I/O-bound processes in a mixed workload environment.

Average Throughput

Average throughput is another important performance measure. The characteristics of average throughput for the systems, **CRRS** and **VRRS**, are shown in Figure 8. In **CRRS**, the average throughput values of I/O-bound and CPU-bound processes are closer to 0.155 and 0.31, respectively, i.e., the average throughput of I/O-bound processes is approximately 50% that of CPU-bound processes in **CRRS**. Where as in **VRRS**, the average throughput of I/O-bound and CPU-bound processes are closer to 0.2 and 0.265, respectively, i.e., the average throughput of I/O-bound processes is approximately 75% that of CPU-bound processes. It is interesting to note that the system throughput characteristics for both the systems, **CRRS** and **VRRS**, are almost overlapped. Now, look at the ratio of average throughput of I/O-bound and CPU-bound processes plotted in Figure 9. The ratio values are closer to 0.5 in the case of **CRRS**; where as in the case of **VRRS**, they are close to 0.73. The Figures 8 and 9 infer that I/O-bound processes receive unfair treatment in **CRRS**, and receive fair deal in **VRRS**.

Disk Utilization

It is seen in the previous sections that I/O-bound processes receive fair deal in **VRRS**. The improvement in performance of I/O-bound processes in **VRRS** has a secondary effect on the utilization of I/O devices (disks). In comparison with a **CRRS**, the corresponding **VRRS** provides a bit preferential treatment to I/O-bound processes by giving them higher priority while they are in the *auxiliary queue*. This leads to better utilization of disks. The Figure 10 shows that disks are relatively more utilized in **VRRS**.

III. SUMMARY

The main contribution of this paper is the introduction of the concept of ***remaining time slice***, and the development of virtual round robin (**VRR**) scheduler. It is also shown that the working principle of the **VRR** scheduler is quite the same as that of conventional round robin (**VRR**) scheduler. The **VRR** scheduler retains all the basic properties of **CRR** scheduler, moreover it behaves evenly and fairly with all types of processes in a mixed environment. If a user tries to cheat the system that uses the **VRR** scheduler with dummy I/O instructions in his/her programs, then the process (an instance of execution of the program) when it moves to the auxiliary queue will get a time slice of "remaining time" of the process. As soon as the "remaining time" of the process is over, it is recycled back to the end of the main queue. Hence, dummy I/O instructions do not help a process to improve its response time. Contrariwise, it is disadvantageous for users to use dummy I/O instructions in their programs.

IV. REFERENCES

[1] I.Adiri and B.Avi-Itzhak, A time-sharing queue, Manag. Sci., Vol.15(11), 1969, p639-657.

[2] I.Adiri, A dynamic time-sharing priority queue, Journal of ACM, Vol.18(4), 1971, p603-610.

[3] U.N.Bhat and R.E.Nance, Busy period analysis of a time-sharing system modeled as a semi-Markov process, Journal of ACM, Vol.18(2), 1971, p221-238.

[4] P.R.Blevins and C.V.Ramamoorthy, Aspects of dynamic adoptive operating systems, IEEE Tran. on Computer, Vol.25(7), 1970, p713-725.

[5] J.E.Burns, M.J.Fischer, P.Jackson, N.A.Lynch and G.L.Peterson, Data requirement for implementation of n-process mutual exclusion using single shared variable, Journal of ACM, Vol.29, 1982, p183-205.

[6] E.G.Coffman, Analysis of two time sharing algorithms designed for limited swapping, Journal of ACM, Vol.15(3), 1968, p341-353.

[7] S.A.Friedberg and G.L.Peterson, An efficient solution to the mutual exclusion problem using unfair and weak semaphores, Information Processing Letters, Vol.25, 1987, p343-347.

[8] S.Haldar and D.K.Subramanian, An efficient solution to the mutual exclusion problem using unfair and weak semaphores, ACM SIG Operating System Review, Vol.22(2), 1988, p60-66.

[9] L.Kleinrock, Analysis of time-shared processor, Naval Res. Linguistics Quart., Vol.2(1), (1964), p59-73.

[10] L.Kleinrock and R.R.Muntz, Processor sharing queueing models of mixed scheduling disciplines for time shared systems, Journal of ACM, Vol.19(3), 1972, 464-482.

[11] L.Kleinrock, Queuing Systems, Volume II: Computer Applications, Wiley, N.York, 1976.

[12] J.McKinney, A Survey of analytical time-sharing models, ACM Computing Surveys, Vol.1(2), (1969), p105-116.

[13] R.E.Nance, U.N.Bhat and B.G.Claybrook, Busy period analysis of a time sharing system:transform inversion, Journal of ACM, Vol.19(3), 1972, p453-463.

[14] J.L.Peterson and A.Silberschatz, Operating system concepts, Addison-Wilesly, 1984.

[15] R.L.Rivest and V.R.Pratt, The mutual exclusion problem for unreliable processes: preliminary report, 17th Annual symposium on Foundation of Computer Science, Houston, Tex, 1976, p1-8.

[16] K.D.Ryder, A heuristic approach to task despatching, IBM system Journal, Vol.9(3), 1970, p189-198.

[17] J.E.Shemer, Some mathematical considerations of time-sharing scheduling algorithms, Journal of ACM, Vol.14(2), 1967, p262-272.

[18] S.Sherman, F.Baskett and J.C.Browne, Trace modeling and analysis of CPU scheduling in a multiprogramming system, Communication of ACM, Vol.15(12), 1972, p1063-1069.

[19] S.E.Madnick and J.J.Donovan, Operating systems, McGraw-Hill Computer Science Series, 13th Printing in 1986 (originally in 1974).

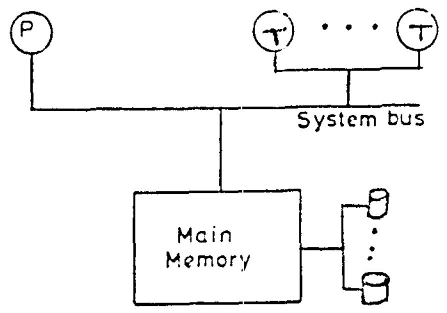

Fig.1 Typical Model of a Computer System.

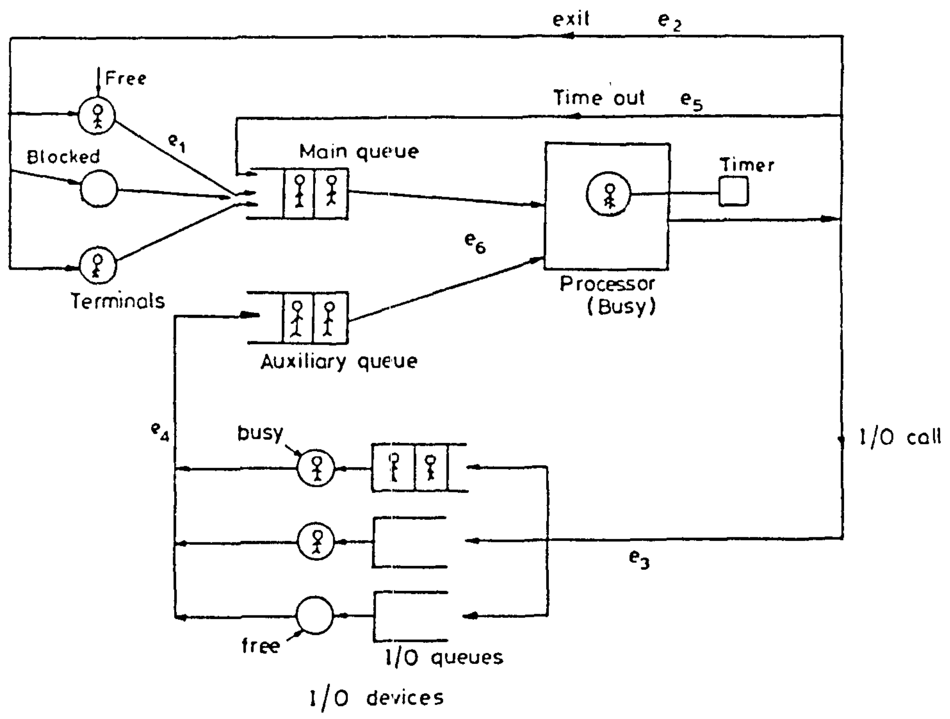

Fig.2 Queuing Model of a Computer System with Virtual Round Scheduler.

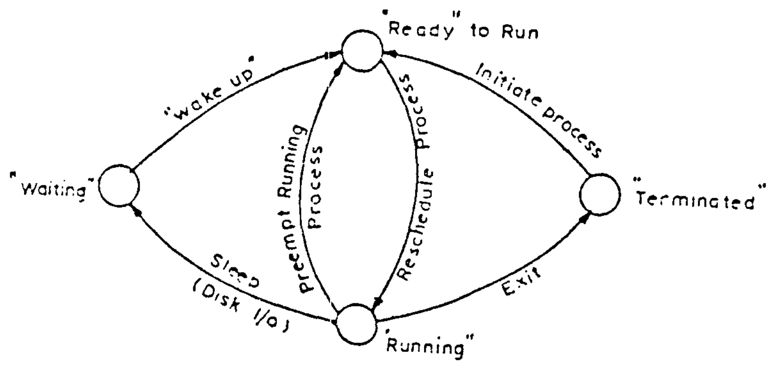

Fig.3 State Transition Diagram of a Process.

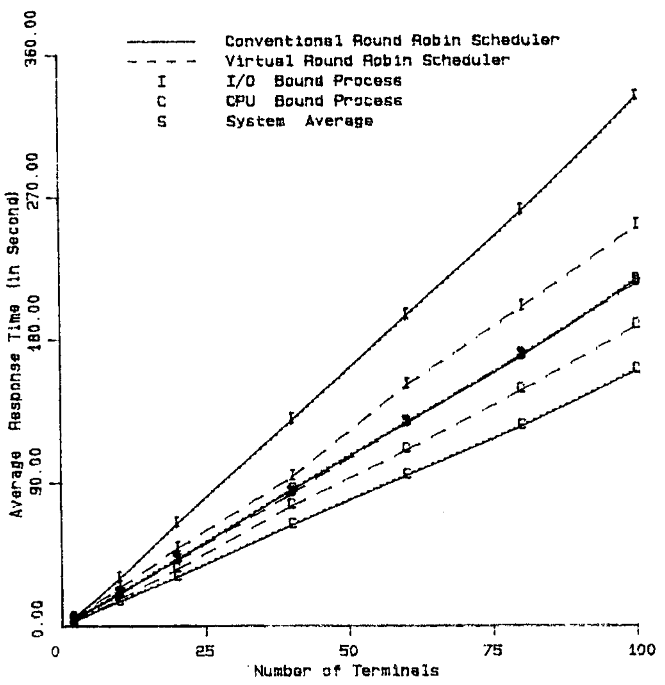

Figure 4. Average Response Time Characteristic.
Standard Environment Characterized by:
50% CPU Bound Processes, 50% I/O Bound Processes,
N.CPU = 1, N.DISK = 4, Q = 400 Msecs, $1/\lambda$ = 1.2 Secs.
$1/\mu$ = 1.6 Secs, $1/\lambda IO$ = 0.4 Sec, $1/\lambda CPU$ = 1.6 Secs

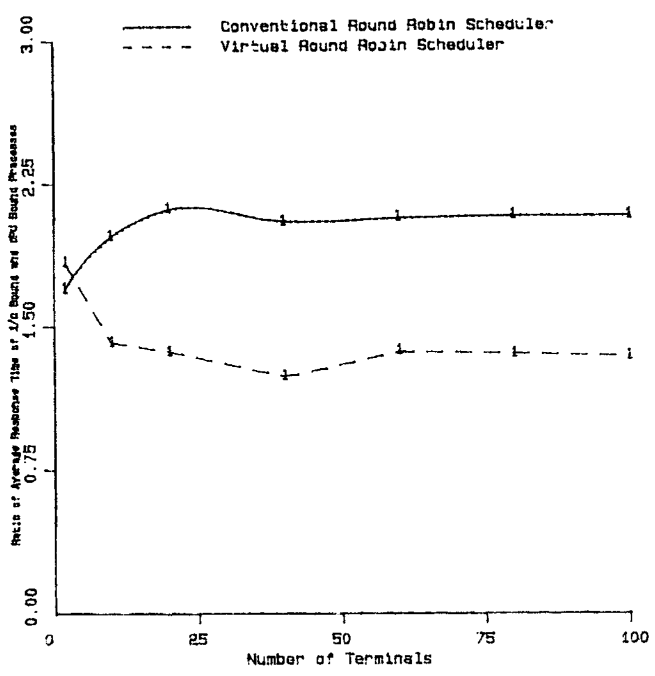

Figure 5. Ratio of Average Response Time of I/O & CPU Bound Processes
Standard Environment Characterized by:
50% CPU Bound Processes, 50% I/O Bound Processes.
N.DISK = 4, Q = 400 Msecs, $1/\lambda$ = 1.2 Secs.
$1/\mu$ = 1.6 Secs, $1/\lambda IO$ = 0.4 Sec, $1/\lambda CPU$ = 1.6 Secs

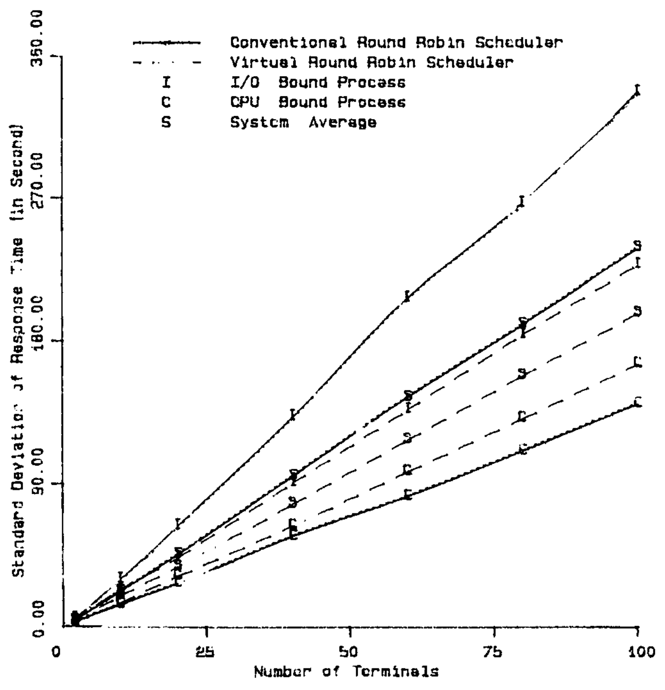

Figure 6. Standard Deviation of Response Time Characteristic.
Standard Environment Characterized by:
50% CPU Bound Processes, 50% I/O Bound Processes,
N.CPU = 1, N.DISK = 4, Q = 400 Msecs, $1/\lambda$ = 1.2 Secs.
$1/\mu$ = 1.6 Secs, $1/\lambda IO$ = 0.4 Sec, $1/\lambda CPU$ = 1.6 Secs

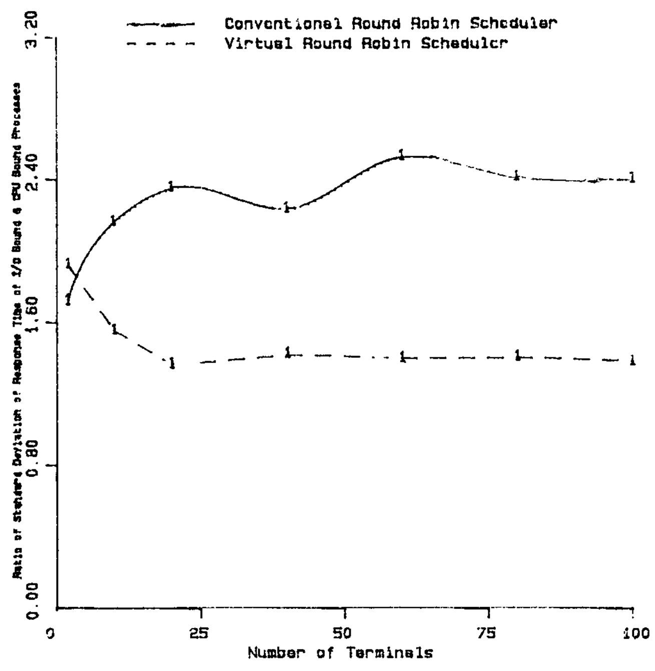

Figure 7. Ratio of STD. DEV. of Resp. Time of I/O & CPU Bound Processes
Standard Environment Characterized by:
50% CPU Bound Processes, 50% I/O Bound Processes,
N.DISK = 4, Q = 400 Msecs, $1/\lambda$ = 1.2 Secs.
$1/\mu$ = 1.6 Secs, $1/\lambda IO$ = 0.4 Sec, $1/\lambda CPU$ = 1.6 Secs

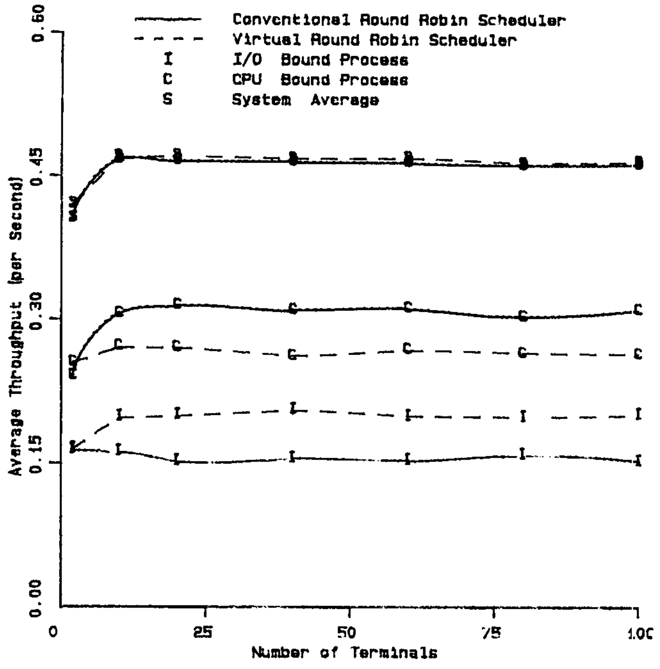

Figure 8. Average Throughput Characteristic.
Standard Environment Characterized by:
50% CPU Bound Processes. 50% I/O Bound Processes.
N.CPU = 1. N.DISK = 4. Q = 400 Msecs. $1/\lambda = 1.2$ Secs.
$1/\mu = 1.6$ Secs. $1/\lambda IO = 0.4$ Sec. $1/\lambda CPU = 1.6$ Secs

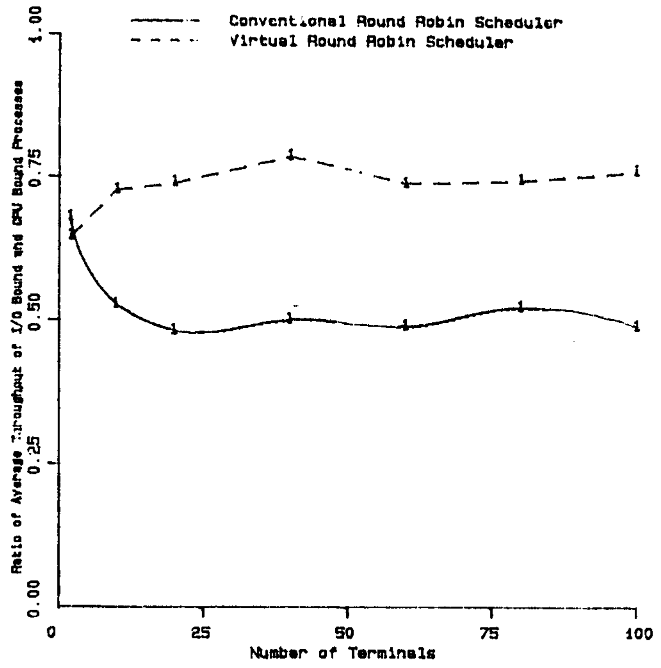

Figure 9. Ratio of Average Throughput of I/O & CPU Bound Processes
Standard Environment Characterized by:
50% CPU Bound Processes. 50% I/O Bound Processes.
N.DISK = 4. Q = 400 Msecs. $1/\lambda = 1.2$ Secs.
$1/\mu = 1.6$ Secs. $1/\lambda IO = 0.4$ Sec. $1/\lambda CPU = 1.6$ Secs

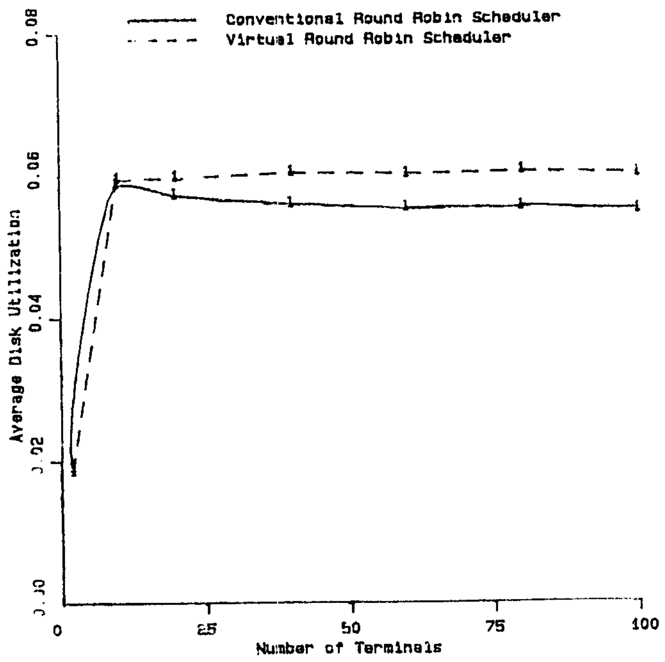

Figure 10. Average Disk Utilization Characteristic.
Standard Environment Characterized by:
50% CPU Bound Processes. 50% I/O Bound Processes.
N.DISK = 4. Q = 400 Msecs. $1/\lambda = 1.2$ Secs.
$1/\mu = 1.6$ Secs. $1/\lambda IO = 0.4$ Sec. $1/\lambda CPU = 1.6$ Secs